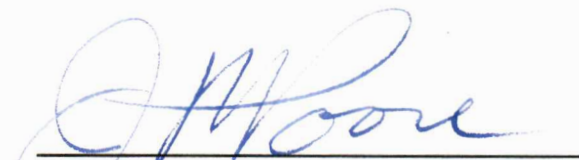
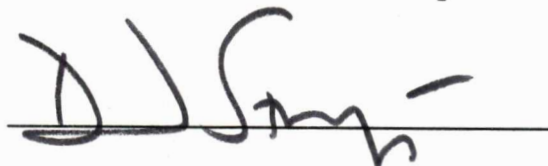


To the Graduate Council:

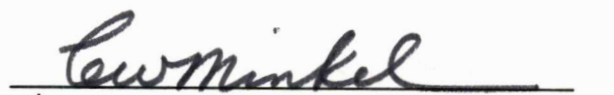
I am submitting herewith a thesis written by Scott Lee Hopkins entitled "A Box Structure Ada Design Language." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.


J. H. Poore, Major Professor

We have read this thesis
and recommend its acceptance:


David C. Mitchell

Accepted for the Council:


Vice Provost
and Dean of the Graduate School

Statement of Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the library shall make it available to borrowers under the rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Scott L. Hopkins

Date 7/24/90

**A BOX STRUCTURE
ADA DESIGN LANGUAGE**

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Scott L. Hopkins

August 1990

ACKNOWLEDGEMENTS

I would like to express appreciation to Dr. Jesse Poore for his guidance and patience in the course of this research. Over the past 18 months, I have learned valuable software engineering principles which will remain with me throughout my professional career. Likewise, thanks to Dr. David Mutchler and Dr. David Straight for serving on my committee.

All students doing research with Dr. Poore (Jim Whittaker, Carmen Trammel, Hailong Mao, David Fuhrer, and Dan Fetzer) have been most supportive. A special thanks to Jim who assisted in the case study by providing his expertise in statistical testing.

Most importantly, I cherish the love and support of my wife, Angela. Surely Solomon spoke of her when he said "the man who finds a wife finds a good thing."

Finally, I express gratitude to my friend Harry Middlebrook, Jr., a man of true wisdom. "I thank my Lord Jesus Christ upon every remembrance of you."

ABSTRACT

Box Structure Methodology is a stepwise system design process which facilitates the application of software engineering principles to systems design. Box Structure Methodology defines a system as an abstract function. It then proceeds, with an eleven step process, to expand that function into a hierarchy of subfunctions each described as a black box, state box, and clear box. The resulting design is expressed in a non-executable language called Box Description Language (BDL). BDL consists of outer syntax for Box Structures (black box, state box, and clear box) and control structures.

The Ada language is executable and was designed to express software engineering principles in systems development. For these reasons an Ada Design Language (ADL) is defined to replace the Box Description Language. Black box to clear box expansion is illustrated for each Ada program unit. The eleven step BDL design process is modified and illustrated.

Finally, a case study is discussed which applied this Box Structure ADL to a small system. A comparison between this case study and a similar study using standard BDL is made.

TABLE OF CONTENTS

1.	Introduction	1
2.	Box Structure Methodology	3
2.1.	Black Box Description	3
2.2.	State Box Description	4
2.3.	Clear Box Description	5
2.4.	Box Structure Design Process	8
2.5.	Box Structure Hierarchy	9
2.6.	Box Structure Design Principles	9
2.6.1.	Referential Transparency	11
2.6.2.	Transaction Closure	11
2.6.3.	State Migration	11
2.6.4.	Common Services	12
3.	Box Description Language	13
3.1.	BDL Black Boxes	13
3.2.	BDL State Boxes	14
3.3.	BDL Clear Boxes	14
3.4.	BDL Control Structures	15
3.4.1.	BDL Sequence Structure	16
3.4.2.	BDL Alternation Structure	16
3.4.3.	BDL Iteration Structure	17
3.4.4.	BDL Concurrent Structure	18

4.	Ada Design Language	19
4.1.	Ada Systems	19
4.2.	Box Structure ADL Goals	20
4.3.	ADL Templates	20
4.3.1.	ADL Procedure Templates	24
4.3.2.	ADL Function Templates	24
4.3.3.	ADL Package Templates	24
4.4.	ADL Control Structures	26
4.4.1.	ADL Sequence Structure	26
4.4.2.	ADL Alternation Structure	27
4.4.3.	ADL Iteration Structure	28
4.5.	ADL Development	29
4.5.1.	Procedure Development	29
4.5.2.	Function Development	31
4.5.3.	Package Development	32
5.	Box Structure Methodology In Ada	36
5.1.	ADL Box Structure Expansion Process	36
5.2.	Illustrating Box Structure ADL	37
5.3.	ADL Diagrams	47
5.3.1.	Control And Data Flow Diagram	47
5.3.2.	Usage Vs. Availability Hierarchies	49
5.3.3.	ADL Structure Diagram	51
6.	An ADL Case Study	56
6.1.	Introduction	56

6.2. Implementing An Edit Window Package	56
6.3. Observations	59
7. Conclusion	60
Bibliography	62
Appendix	65
Vita	92

LIST OF FIGURES

Figure	Page
1. Black box diagram	4
2. State box diagram	5
3. Clear box sequence structure	6
4. Clear box alternation structure	6
5. Clear box iteration structure	7
6. Clear box concurrent structure	7
7. Box structure hierarchy	10
8. The case clear box structure	17
9. Box Structure ADL Template	21
10. Clear box for Swap	30
11. Clear box for Max	32
12. Clear box for Swap_Max	35
13. Control and data flow diagram of Menu	48
14. ADL usage hierarchy	50
15. ADL availability hierarchy	50
16. Subprogram invocation of external black boxes .	52
17. Subprogram invocation of internal black boxes .	52
18. Package specification of black boxes	52
19. Black box for Menu	54
20. Clear box for Menu	54
21. Clear box for Menu	55
22. Structure diagram for Screen5_F6	58

Chapter 1

INTRODUCTION

The Box Structure Method (BSM) is a well established top-down functional design method that facilitates intellectual control over large projects. It takes the view that a system, at its highest level, can be described as a function [15]. This abstract system function is then redefined through a process of stepwise refinement, eventually expanding the function into its language specific implementation. The repeated stepwise refinement defines functions at each level in terms of increasingly simpler functions.

The Box Structure Method was developed by Harlan D. Mills [8] to meet the need for a formal, incremental process for design and presentation of large information systems. The resulting methodology is a top-down process allowing a development team to take small, verifiable steps measuring progress toward system completion.

While systems development is very much a creative process of invention and discovery, it is also true that most advances in software engineering have had a theoretical, mathematical basis (e.g. structured programming). The Box Structure Method is effective as a design methodology for much the same reason.

BSM uses mathematical ideas as the basis for system design. These designs are expressed in a pseudo code called Box Description Language (BDL). There are no compilers or interpreters for BDL. The design must be transcribed into an executable language.

The Ada language was designed to facilitate the application of software engineering concepts during systems development [1]. These software engineering concepts include information hiding, data abstraction, object oriented design, and reusability to name a few. It has been an objective of the Software Technology for Adaptable Reliable Systems (STARS) community to develop and use a design language that is executable Ada [14].

Ada and BSM share many software engineering objectives. BSM is strong in design, but is not executable. Ada is weaker than BSM for design, but is executable on many computers. The development of a Box Structure ADL would allow the compilation and execution of systems developed by BSM and expressed in the Box Structure ADL.

Chapter 2

BOX STRUCTURE METHODOLOGY

The Box Structure Method is a way of describing a system (or subsystem) as a hierarchy of functions and data abstractions. Each such abstraction is composed of a black box, state box, and clear box.

2.1. Black Box Description

The black box describes the system by mapping each stimulus to its appropriate response. Hence, the black box is an external functional description of system behavior. As its name suggests, the black box does not address details of internal behavior and data representation, thus eliciting a more exhaustive analysis of stimuli than might otherwise occur.

Let S be the set of possible stimuli, S^* the set of sequences of S , and R the set of possible responses. The black box is a function

$$f: S^* \rightarrow R$$

mapping S^* to R [9]. All black box responses are uniquely determined by the stimulus history, i.e. sequences. A black box diagram is shown in Figure 1.

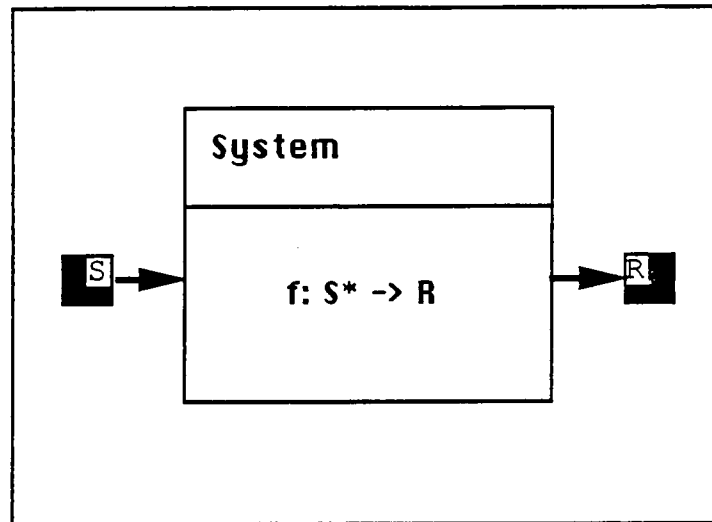


Figure 1. Black box diagram (Mills, 1987).

2.2. State Box Description

The state box description of a function differs from the black box description by retaining sufficient state data to capture essential stimulus history. Furthermore, transaction rules, referred to as "the machine" in Figure 2, are used to describe the functionality of the state box. The state box is modeled after a state machine (from which it inherits its name), the difference is that the state box can take advantage of a hierarchy of data abstractions for representing its state. These data abstractions facilitate the storing of stimuli to respond to the effects of previous stimuli.

Like the black box, the state box has a functional description. Let T be the set of all states and T^* the state histories. The state box is a function described by an

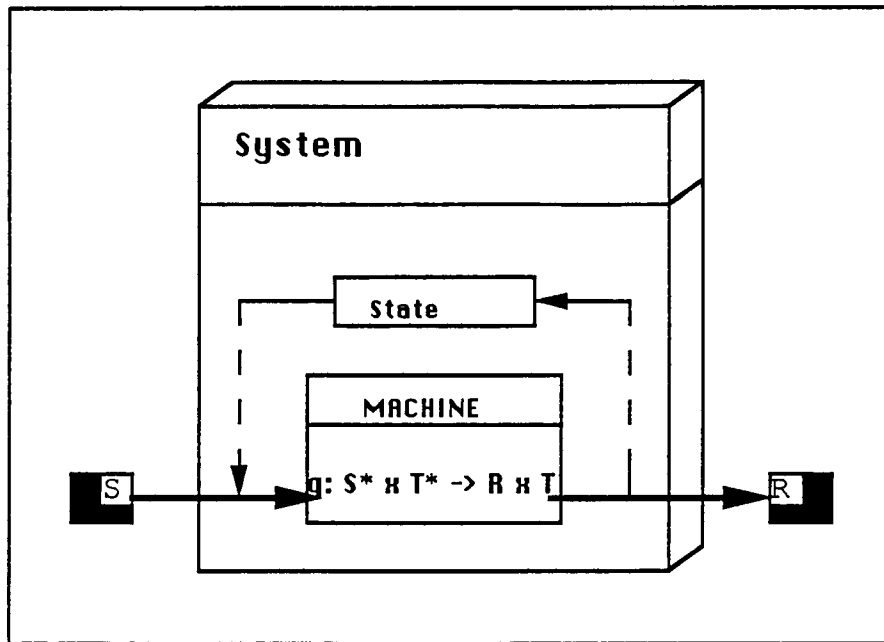


Figure 2. State box diagram (Mills, 1987).

initial state t and function

$$g: S^* \times T^* \rightarrow R \times T$$

As the function g reveals, a state box takes a stimulus and old state and produces a response and new state.

2.3. Clear Box Description

The purpose of the clear box is to prescribe how to implement the system behavior. The clear box is a procedural description which incorporates a basis set of control structures, e.g. sequence, alternation, iteration, and concurrent, to describe the processing of the stimulus and state as shown in Figures 3-6. These structures direct the control flow to transactions invented for this process description. During this expansion phase, new black box

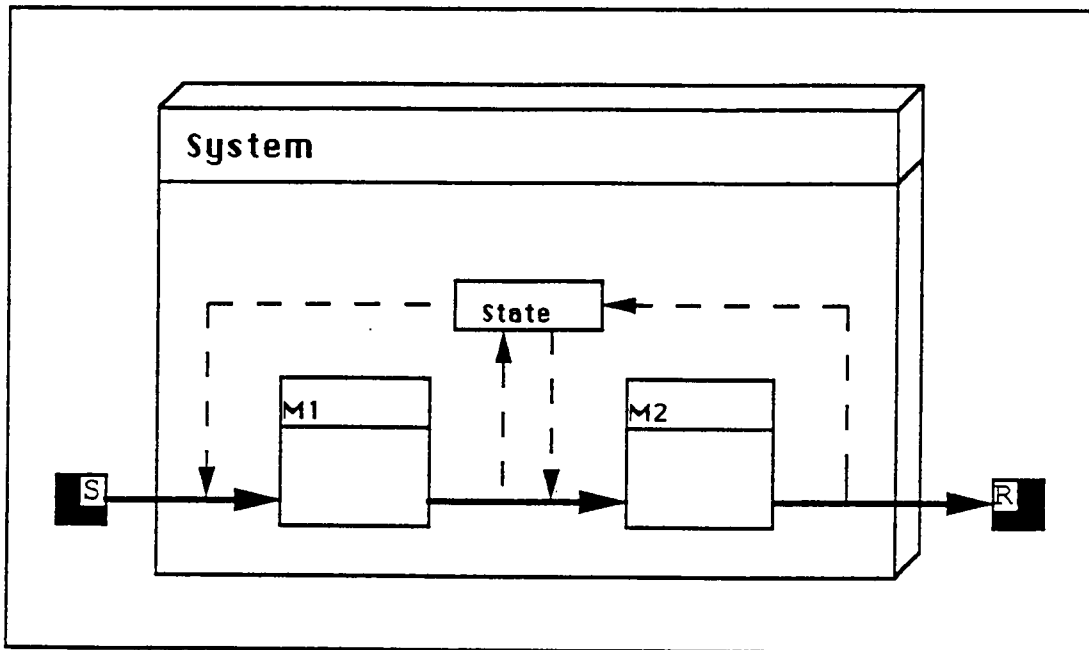


Figure 3. Clear box sequence structure (Mills, 1987).

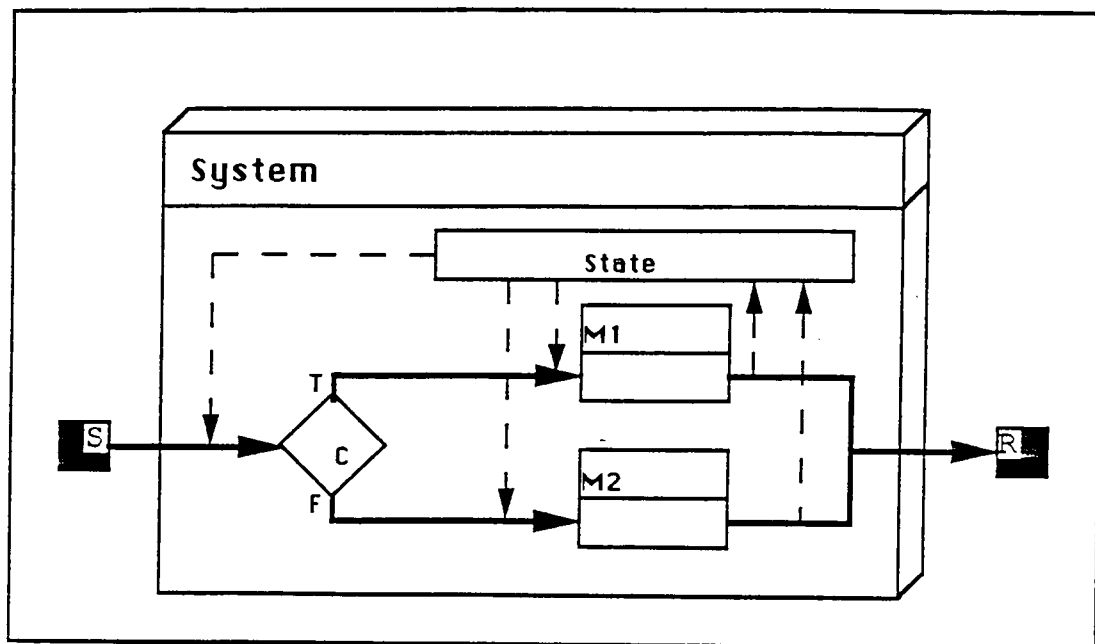


Figure 4. Clear box alternation structure (Mills, 1987).

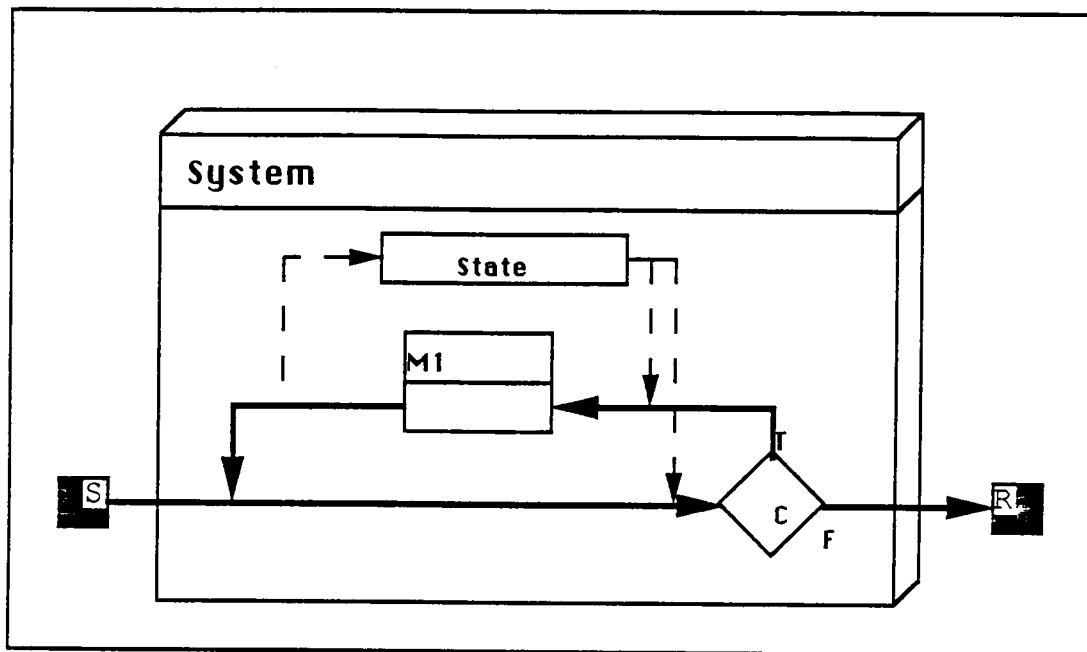


Figure 5. Clear box iteration structure (Mills, 1987).

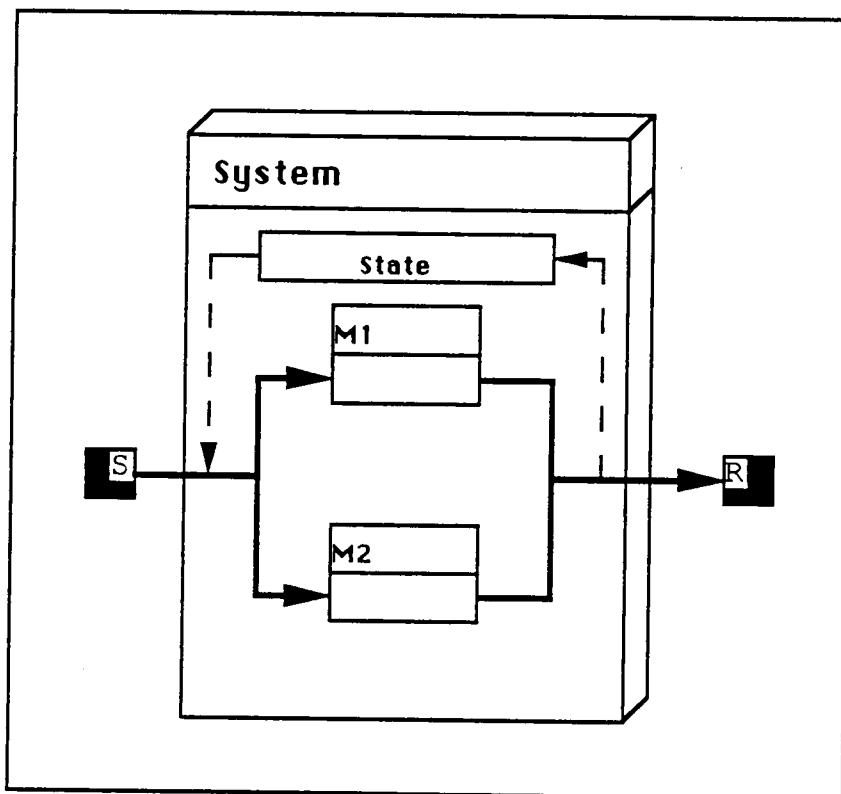


Figure 6. Clear box concurrent structure (Mills, 1987).

abstractions may be invented. The expansion is then repeated for new or remaining black boxes.

2.4. Box Structure Design Process

Box structure system design is a stepwise refinement and verification process for converting specifications into hierarchical system designs. Mills [9] outlines this process for black box to clear box expansion as follows:

Define the black box

- (1) Define black-box stimuli
Determine all possible stimuli for the black box.
- (2) Define black-box behavior
For each possible stimulus, determine its complete response in terms of its stimulus history.

Design the state box

- (3) Discover state data requirements
For each response to be calculated, encapsulate its stimulus history into a state data requirement.
- (4) Define the state
Select a subset of the required state data items to encapsulate stimulus histories.
- (5) Design the state box
For the selected state, determine the internal black box required for the state box.
- (6) Verify the state box
Verify the correctness of the state box with respect to the required black-box behavior.

Design the clear box

- (7) Discover the state data accesses
For each item of state data and each possible stimulus, determine all possible accesses of the item.
- (8) Define data abstractions
Organize state data into data abstractions for effective access.
- (9) Design the clear box

Define sequential or concurrent uses of the data abstractions defined to replace the internal black box of the state box.

- (10) Verify the clear box
Verify the correctness of the clear box with respect to the state-box behavior.

Continue the process

- (11) Repeat stepwise expansion until design completion
For each new data abstraction, repeat steps 1-10 until suitable data and program specifications are reached.

2.5. Box Structure Hierarchy

A box structure hierarchy represents the order of usage of a box structure system. These hierarchies prove to be useful in the analysis and management of systems during development. Box structure hierarchies (Figure 7), reflecting the divide and conquer strategy of BSM, provide for the repeated application of the expansion rules to smaller and smaller subsystems.

2.6. Box Structure Design Principles

Mills [7] presents four design principles pertinent to the successful design of any system. Box Structures provides a structured iterative framework for systematic application of these principles. These principles are referential transparency, transaction closure, state migration, and common services.

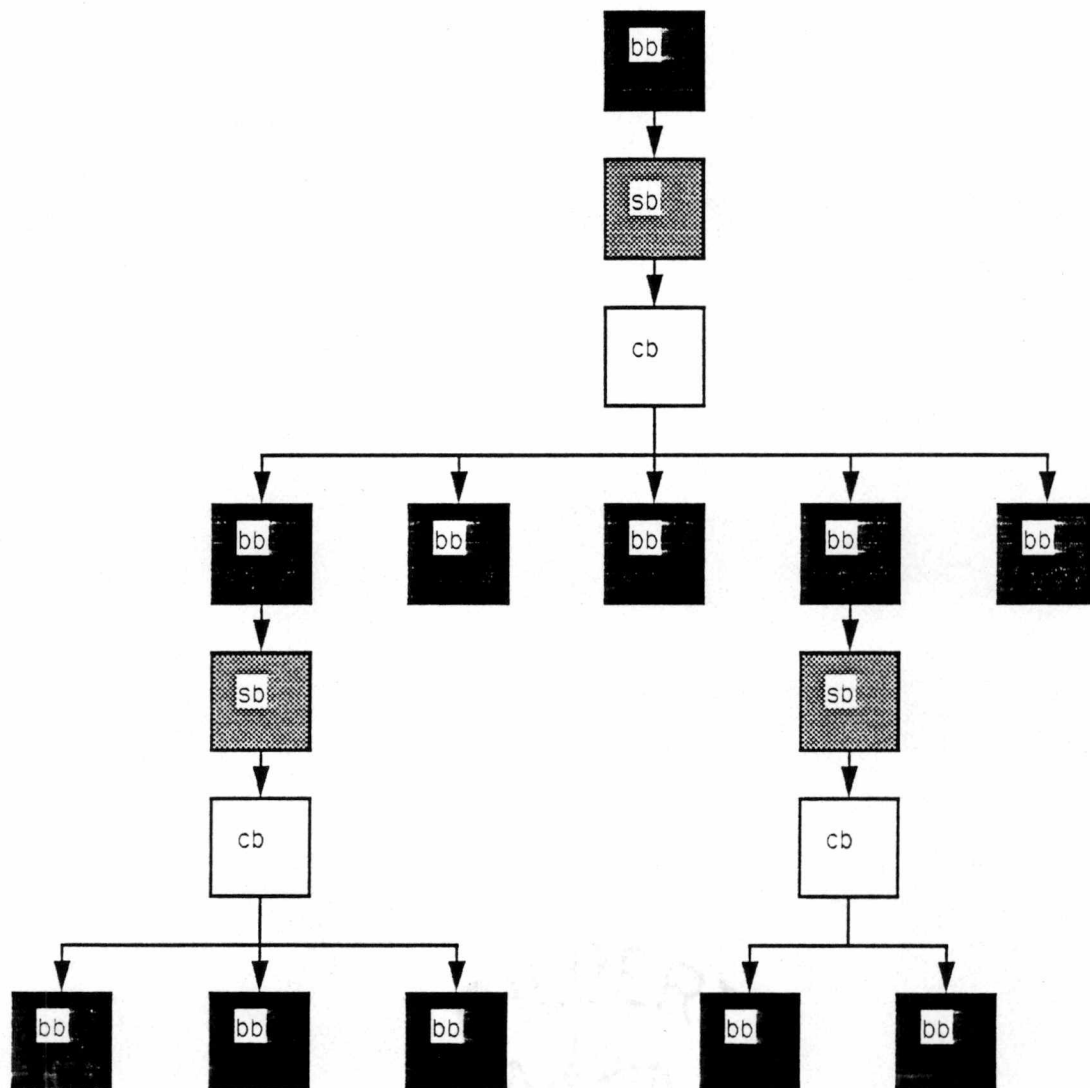


Figure 7. Box structure hierarchy.

2.6.1. Referential Transparency

The *Principle of Referential Transparency* asserts that any system part designated for development first is specified, that is, all information necessary to design and implement that part is established prior to its delegation. No further communication is required. Therefore, responsibility can likewise be transferred to the assigned development team. Referential transparency is an important consideration during the discovery of data abstractions.

2.6.2. Transaction Closure

The *Principle of Transaction Closure* asserts that the transactions accompanying a system should be sufficient to preserve its state data, and its state data should suffice for completion of its transactions. The declaration of transactions and state data is an iterative, spiral process of system specification. Mills explains that,

The iteration begins with the transactions for the primary users, and the state data needed for those transactions, then considers the transactions required for the acquisition and preservation of those state data, then identifies the state data needed for those transactions and so on. Eventually, no more transactions will be required in an iteration, and transaction closure will have been achieved. [7]

2.6.3. State Migration

The *Principle of State Migration* asserts that state data should be

migrated down the system hierarchy as far as possible to still allow visibility to state accessing transactions, and not require duplicate state updates. Given such a strategy, discussions concerning state data structures as well as their usage are postponed, often preventing premature decisions from being incorporated into the design.

2.6.4. Common Services

The *Principle of Common Services* asserts that productivity and system reliability are greatly enhanced when the potential of a part to provide service to multiple other parts is recognized. All common services are to be modules in the sense defined and described by Parnas [12].

Chapter 3

BOX DESCRIPTION LANGUAGE

The Box Description Language [8] (BDL) was prescribed to give an outer syntax to the formal process of BSM. As Mills introduced it,

Box structures provides a theoretical foundation for information systems development and use, and Box Description Language, or BDL, provides a corresponding language for recording, communicating, and analyzing box structures among developers, users, and managers of information systems. [8]

The outer syntax defines consistent structure for all boxes and assures that transition rules conform to the control structures in the basis. Meanwhile, inner syntax can be chosen as any notation deemed suitable for the problem at hand. Once the box structure expansions have been finished and the target language selected, then a transliteration into that target language must be performed.

3.1. BDL Black Boxes

The BDL syntax for defining a black box is

```
define BB <BB name>
  stimulus
    <stimulus name>:<type>
  response
    <response name>:<type>
  transition
    <BB transition>
```

define BB, stimulus, response, and transition are keywords

used as outer syntax. The remainder of the black box is user chosen notation. **stimulus** denotes the list of stimuli, named <stimulus name>, each given an associated <type>. Likewise, **response** identifies each <response name> and <type>. Each black box describes its behavior by mapping all stimuli to their responses by means of <BB transition> rules.

3.2. BDL State Boxes

The state box relies upon a state to capture the stimulus history according to BDL syntax

```
define SB <SB name>
  stimulus
    <stimulus name>:<type>
  response
    <response name>:<type>
  state
    <state data>
  transition
    <SB transition>
```

This box differs in only two ways from the black box. Outer syntax **state** denotes the <state data> necessary to represent the stimulus history. Secondly, the <SB transition> rules reflect this effort with state data accesses.

3.3. BDL Clear Boxes

The clear box will incorporate lower level black boxes to accomplish the state box. That is, these lower level black boxes will work together to maintain the state data and generate correct responses. The BDL syntax for the clear box is

```

define CB <CB name>
  stimulus
    <stimulus name>:<type>
  response
    <response name>:<type>
  state
    <state data>
  machine
    data
      <procedure data>
    proc
      <procedure>
    corp

```

proc and **corp** are keywords delimiting the <procedure> for the **machine**. This procedure involves control structures which control the invocation of lower level black boxes and the flow of data to them. The origin of this data can be stimuli, state data, or <procedure data>. <procedure data>, denoted by keyword **data**, is local, working data used by the machine procedure. <stimulus name>, <response name>, <type>, and <state data> have the same meanings as before. Additional outer syntax is introduced for describing the clear box procedure.

3.4. BDL Control Structures

BDL control structures allow the specification of control and data flow through a set of black boxes. These structures represent single entry, single exit control structures [8]. Furthermore, they are used in the procedural description of the black box, state box, and clear box.

3.4.1. BDL Sequence Structure

A clear box may accomplish its transition by invoking a sequence of black boxes, say BB1 and BB2, as follows:

```
do
  BB1;
  BB2
od
```

do and **od** are keywords providing outer syntax. The response of BB1 becomes the stimulus to BB2. In addition BB1 may create a new state for BB2 (see Figure 3).

3.4.2. BDL Alternation Structure

With an alternation structure (Figure 4), a conditional test C directs the stimuli to either BB1 or BB2

```
if
  C
then
  BB1
else
  BB2
fi;
```

While the number of black boxes an alternation structure has to select from may vary, at most one is chosen to receive the stimuli. In fact, the case structure

```
case
  E
part (value1)
  BB1
part (value2)
  BB2
part (value3)
  BB3
esac
```

is a generalization of the alternation structure (Figure 8). **case** and **esac** delimit the case statement. **part** defines the

procedure that will be executed should E evaluate to the corresponding value.

3.4.3. BDL Iteration Structure

An iteration structure (Figure 5) repeatedly invokes a black box until a condition C becomes false. The BDL syntax is

```
while
  C
do
  BB1
od;
```

while, **do**, and **od** provide outer syntax. Note that BB1 may be invoked zero or more times.

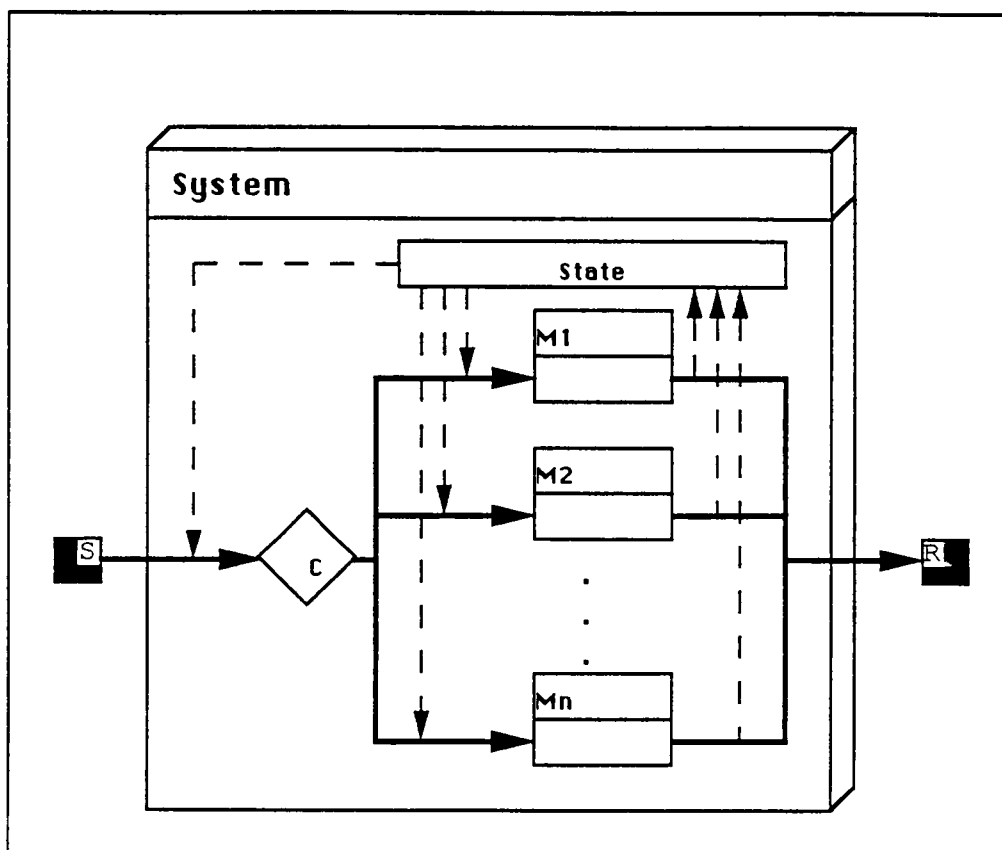


Figure 8. The case clear box structure.

3.4.4. BDL Concurrent Structure

Concurrent control structures (Figure 6) concurrently execute black boxes. The BDL syntax for this structure is

```
con  
    BB1;  
    BB2  
noc;
```

with keywords **con** and **noc** giving the outer syntax for the concurrent execution of BB1 and BB2.

Chapter 4

ADA DESIGN LANGUAGE

Interest in using the box structure methodology has been expressed by users of the Ada language [2,3]. The development of an Ada system is a process of discovery and invention of data abstractions and data objects. In theory and in practice, BSM provides a stepwise framework for such analysis.

BDL was invented to provide a text for the Box Structure Methodology. When the target language is known in advance, the time consuming and error prone process of transliteration from BDL to that language can be avoided. An Ada Design Language (ADL) can replace BDL when Ada is the target language.

4.1. Ada Systems

An Ada program is an Ada system, however Ada systems are not limited to Ada programs. An Ada system is,

... a collection of Ada programs and system files such that the system files are accessed and altered only by the Ada programs of the Ada system. An Ada system cannot be described in executable Ada, even though all of its programs can. But Ada has a construct, the Ada package, that can be used to define an Ada system precisely. It will contain the names of all Ada programs of the system... [6]

Comments can play a big role in describing Ada systems as box

structures.

Comments can be used to describe functional behavior of each program and the meanings of the system files in the system package. This description can reach down to every user stimulus possible in each program and the corresponding response [6].

4.2. Box Structure ADL Goals

By use of Box Structure ADL we hope to achieve the following goals:

- (1) The ADL will be Ada as described by ANSI/MIL-STD-1815A [3], not a restricted subset.
- (2) Provide a language for expressing the hierarchical expansion of box structures.
- (3) The box structure ADL should support the reuse of code.
- (4) Allow the entire design (black box, state box, and clear box) to be compilable by a compiler that correctly implements ANSI/MIL-STD-1815A. Hence, the source code should document the entire development process, the black box and state box will rely heavily upon comments while the clear box will be primarily executable Ada.
- (5) The black box, state box, and clear box should combine to make one compilable unit.
- (6) A standard format will be defined for outer syntax in order to enhance its accommodation to automated tools for design and verification [4].

4.3. ADL Templates

Ada Box Structures uses the following ADL templates for procedures, functions, and packages to replace the BDL frameworks (Figure 9).

Procedure Specification

```
-----  
-- Black Box Requirement for BOX_NAME  
--  
-- <Author>  
-- <Company>  
-- <Designed>  
-- <Modified>  
-----  
with {simple_unit_name}  
procedure BOX_NAME  
(  
-----  
-- Stimuli and Responses for BOX_NAME  
--  
-- [declarative_part]  
-----  
-- Transactions for BOX_NAME  
--  
-----  
);
```

Procedure Body

```
with {simple_unit_name}  
procedure BOX_NAME  
(  
-----  
-- Stimuli and Responses for BOX_NAME  
--  
-- [declarative_part]  
-----  
)  
is  
-----  
-- State for BOX_NAME  
--  
-- [declarative_part] (for types and objects)  
-----  
-- State Box Transactions for BOX_NAME  
--  
-----  
-- Clear Box Transactions for BOX_NAME  
-- [declarative_part]  
begin  
-- [sequence_of_statements]  
-----  
end BOX_NAME;
```

Figure 9. Box Structure ADL Template.

Function Specification

-- **Black Box Requirement** for BOX_NAME

--
-- <Author>
-- <Company>
-- <Designed>
-- <Modified>

With {simple_unit_name}
function BOX_NAME

(

-- **Stimuli and Responses** for BOX_NAME
-- [declarative_part]

-- **Transactions** for BOX_NAME
--

)
return RETURN_TYPE;

Function Body

with {simple_unit_name}
function BOX_NAME

(

-- **Stimuli and Responses** for BOX_NAME
-- [declarative_part]

)
return RETURN_TYPE
is

-- **State** for BOX_NAME
--
-- [declarative_part] (for types and objects)

-- **State Box Transactions** for BOX_NAME
--

-- **Clear Box Transactions** for BOX_NAME
-- [declarative_part] (for types and objects)

begin
-- [sequence_of_statements]

end BOX_NAME;

Figure 9. (continued)

Package Specification

```
-----  
-- Black Box Requirement for BOX_NAME  
--  
-- <Author>  
-- <Company>  
-- <Designed>  
-- <Modified>  
-----  
with {unit_simple_name}  
package BOX_NAME  
is  
    -----  
    -- Stimuli, Responses, and Transactions for BOX_NAME  
    --  
    -- {basic_declarative_item}  
    -----  
end BOX_NAME;
```

Package Body

```
with {unit_simple_name}  
package body BOX_NAME  
is  
    -----  
    -- State for BOX_NAME  
    --  
    -- [declarative_part] (for state types and objects)  
    -----  
    -- State Box Transactions for BOX_NAME  
    --  
    -----  
  
    -----  
    -- Clear Box Transactions for BOX_NAME  
    --  
    -- [sequence_of_black_boxes]  
    -----  
end BOX_NAME;
```

Figure 9. (continued)

4.3.1. ADL Procedure Templates

The procedure specification template begins with a header used for configuration and version control. This may, of course, suit the needs of the working group. The stimuli and responses will be Ada arguments if they are not commented. If no stimuli or responses exist then the parentheses must be removed. The transactions are commented, therefore they may be expressed in notation suitable to the application.

The procedure body template retains the same stimuli and responses, unfolding the black box transactions in the state and clear boxes. The state box transactions are commented transaction rules reflecting state access. The clear box will be primarily Ada statements constructed with ADL control structures (see section 4.4).

4.3.2. ADL Function Templates

The templates for the function specification and the function body are similar to the procedure templates but include a return type. Formally, this return type will be the only Ada defined response.

4.3.3. ADL Package Templates

The package specification template begins with a header. The Ada defined stimulus to a package specification is a subprogram (function or procedure) name. Any stimuli to such

a subprogram are also considered stimuli to the package. The Ada defined response from a package comes through a function name. Again the responses from the subprograms of the package specification will also be package responses. In addition, other commented stimuli and responses may be specified.

In the package body template, the state box allows for the declaration of the state and its associated data types. The state transition rules are the rules in the package specifications augmented to reflect state access. This reveals how the state captures the stimulus history to provide the appropriate responses. The state transition rules should all be commented, otherwise, multiple declarations of these black boxes would exist.

The package clear box consists of the transactions of the packages and subprograms in the package specification, as well as the transactions of the data abstractions that will be extracted from them. Transactions of the clear box must be executable Ada statements.

A package receives a stimulus stream $(A, A_1, A_2, \dots, A_n)$ such that A is a subprogram name and $A_1 \dots A_n$ are the stimuli to this subprogram. The package behavior can be viewed as examining the first stimulus element (the subprogram name) to determine the black box to which to pass the remaining

stimuli. For this reason, the package clear box diagram is the same as the **case** clear box diagram of Figure 8.

It has been observed [11] that because the package is a multi-function module, it has more the features of a Parnas module [12] than a subprogram. While the package is like the **case** and **if** statements in allowing a stimulus to determine which black box will be executed, it differs by (1) permitting greater flexibility in what constitutes a stimulus or a response and (2) retaining internal state data between invocations. Also, each template (black box, state box, clear box) forms one compilable library unit.

4.4. ADL Control Structures

Control structures direct the invocation of black boxes, the use of transactions, and the flow of information. As in BDL, these structures represent single entry, single exit control structures [8]. In the following discussion of ADL control structures, any sequence of procedural statements similarly could be a black box invocation.

4.4.1. ADL Sequence Structure

For each black box, the sequence control structure is part of the template. The statement

```
begin
  S1;
  S2
end;
```

is delimited by keywords **begin** and **end**, and is outer syntax for successive invocation of procedural statements.

4.4.2. ADL Alternation Structure

There are two alternation control structures in Ada. The first uses keywords **if**, **then**, **elsif**, **else**, and **end if**. There may be zero or one occurrence of **else** and zero or more occurrences of **elsif**. Each **if** and **elsif** has an associated condition used to determine which sequence of procedural statements will be chosen. One of many configurations might be

```
if
  C
then
  S1
else
  S2
end if;
```

The second alternation structure is comprised of keywords **case**, **is**, **when**, **end case**, and the symbol **=>**. The statement

```
case
  E
is
when
  (value1)
=>
  S1
when
  (value2)
=>
  S2
end case;
```

compares each (value), in order of occurrence, to the expression E until a match is found. The associated sequence

of statements is then executed. Each disjoint block of procedural statements is introduced by **when** (value) => that compares the (value) to the expression. Ada requires a match between E and one of the listed values.

4.4.3. ADL Iteration Structure

There are two basic control structures for iteration. The keywords **while**, **loop**, and **end loop** delimit a conditional loop as follows,

```
while
  C
loop
  S1
end loop;
```

The procedural statements will be executed as long as C is true.

Likewise **for**, **loop**, and **end loop** are also keywords delimiting a loop. For instance,

```
for
  R
loop
  S1
end loop;
```

iterates across a discrete range R, executing S1 once for each value in this range.

The Ada language includes a free form loop with user defined exits. We recommend that this construct not be used because of the difficulties it poses both in correct use and

verification.

4.5. ADL Development

An illustration of black box to clear box development for each template is presented next. The process of constructing Ada systems by multi-leveled abstraction and development of black boxes will be the topic of chapter five.

4.5.1. Procedure Development

A procedure accepts multiple stimuli and provides multiple responses. Suppose, for instance, a black box will accept a sequence of two integers X, Y as stimuli and generate Y, X as the response. The procedure template is

```
procedure Swap
(
  -----
  -- Stimuli and Responses for Swap
  --
  X : in out Integer;
  Y : in out Integer
  -----
  -- Transactions for Swap
  --
  -- swap the values of X and Y.
  -----
);
```

The argument list shows that X and Y are stimuli by defining their modes as "in" and that they are also responses by defining their modes as "out". The function of the box is to exchange the values X and Y .

Since no state data is needed, the clear box may be developed

directly.

```
procedure Swap
(
-----
-- Stimuli and Responses for Swap
--
X : in out Integer;
Y : in out Integer
-----
)
is
-----
-- Clear Box Transactions for Swap
--
Temp : Integer;
begin
Temp := X;
X := Y;
Y := Temp;
-----
end Swap;
```

In the clear box expansion a sequence control structure is used to accomplish the black box transition rule. The discovery of a local variable Temp resulted from the clear box expansion. The clear box is diagrammed in Figure 10.

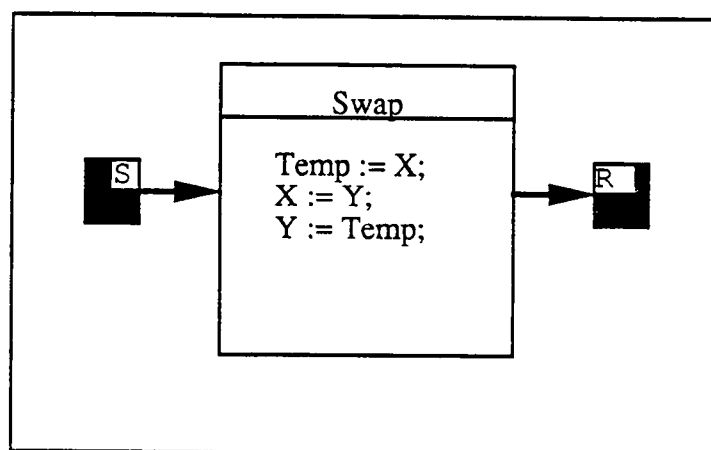


Figure 10. Clear box for *Swap*.

4.5.2. Function Development

Functions are best suited for single response subsystems. For example, the black box *Max* defines a function which returns the maximum of two integers.

```
function Max
(
-----
-- Stimuli and Responses for Max
--
X : in    Integer;
Y : in    Integer
-----
-- Transactions for Max
--
-- if
--   X is greater than Y
-- then
--   return X
-- else
--   return Y
-- end if;
-----
)
return Integer;
```

Again, no state data is required and the clear box expansion is straight forward. As this function body illustrates, only the test "X is greater than Y" required editing into executable Ada to complete the clear box. Figure 11 shows the clear box diagram of *Max*.

```
function Max
(
-----
-- Stimuli and Responses for Max
--
X : in    Integer;
Y : in    Integer
-----
)
return Integer
is
-----
```

```

-- Clear Box Transactions for Max
--
begin
  if
    X > Y
  then
    return X;
  else
    return Y;
  end if;
-----
end Max;

```

4.5.3. Package Development

Suppose a package is to contain these two subprograms, *Swap* and *Max*. The subprogram names serve as the package stimuli. Naming the package *Swap_Max* gives

```

package Swap_Max
is

```

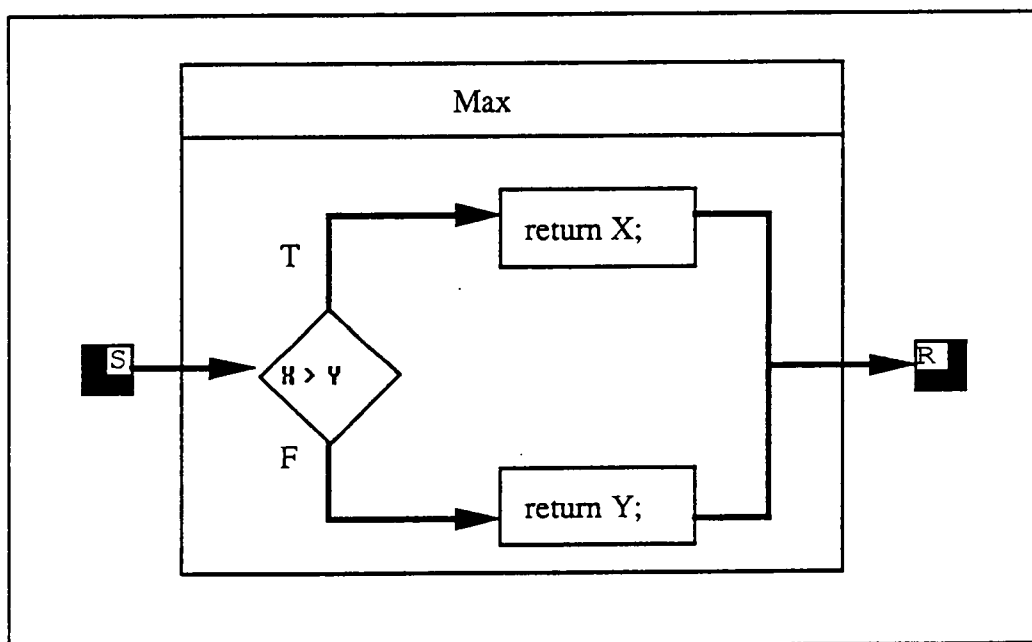


Figure 11. Clear box for *Max*.

```

-----
-- Stimuli, Responses, and Transactions for Swap_Max
--
procedure Swap
(
-----
-- Stimuli and Responses for Swap
--
X : in out Integer;
Y : in out Integer
-----
-- Transactions for Swap
--
-- swap the values of X and Y.
-----
);

function Max
(
-----
-- Stimuli and Responses for Max
--
X : in Integer;
Y : in Integer
-----
-- Transactions for Max
--
-- if
--   X is greater than Y
-- then
--   return X
-- else
--   return Y
-- end if;
-----
)
return Integer;
-----
end Swap_Max;

```

The black box *Swap_Max* contains two black boxes *Swap* and *Max*. Observe that actually the package stimuli are in two parts: (1) the subprogram names and, (2) the stimuli to these subprograms.

The package body encapsulates the bodies of the corresponding

subprograms. Figure 12 depicts this clear box.

```
package body Swap_Max
is
```

```
-----
-- Clear Box Transactions for Swap_Max
--
```

```
procedure Swap
```

```
(
```

```
-----
-- Stimuli and Responses for Swap
--
```

```
X : in out Integer;
```

```
Y : in out Integer
-----
```

```
)
```

```
is
```

```
-----
-- Clear Box Transactions for Swap
--
```

```
Temp : Integer;
```

```
begin
```

```
Temp := X;
```

```
X := Y;
```

```
Y := Temp;
-----
```

```
end Swap;
```

```
function Max
```

```
(
```

```
-----
-- Stimuli and Responses for Max
--
```

```
X : in Integer;
```

```
Y : in Integer
-----
```

```
)
```

```
return Integer
```

```
is
```

```
-----
-- Clear Box Transactions for Max
--
```

```
begin
```

```
if
```

```
X > Y
```

```
then
```

```
return X;
```

```
else
```

```
return Y;
```

```
end if;
-----
```

```
end Max;
```

end Swap_Max;

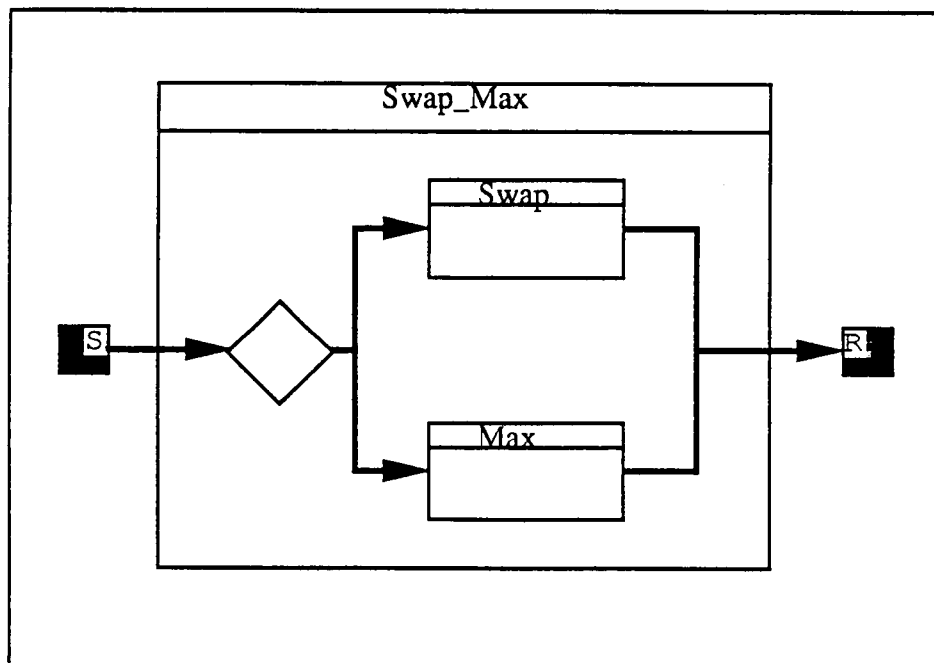


Figure 12. Clear box for *Swap_Max*.

Chapter 5

BOX STRUCTURE METHODOLOGY IN ADA

5.1. ADL Box Structure Expansion Process

Ada packages complicate the design process, but offer a facility for accomplishing many box structure goals. The rules for stepwise refinement are restated with specific reference to Ada as follows.

Define the black box

(1) Define black-box stimuli

Determine all possible stimuli for the procedure, function, or package. If the black box is a package, apply step 1 to each procedure and function defined as a stimulus.

(2) Define black box behavior

For each possible stimulus, determine its complete response in terms of its stimulus history.

Design the state box

(3) Discover state data requirements

For each black box transaction, encapsulate stimulus history into a state data requirement.

(4) Define the state

Determine sufficient state data to capture the stimulus history. If the black box is a package then begin expanding the package body by defining body templates for the subprogram specifications and defining the package state. After defining the package state, apply steps 3-10 to each box in the package.

(5) Design the state box

Define the state transaction rules. Modify the black box transactions to reflect each rule's access to state data.

(6) Verify the state box

Verify the correctness of the state box with respect to the required black-box behavior.

Design the clear box

(7) Design data abstractions

Organize state data into data abstractions for effective access.

(8) Design the clear box

Analyzing the state box transaction rules, convert blocks of one or more lines to either (a) a sequence of one or more Ada statements or (b) a procedure, function, or package by making this block of transaction rules the black box transaction rules for the new subprogram or package, replacing them with an associated calling sequence.

(9) Verify the clear box

Verify the correctness of the clear box with respect to the state box behavior.

Continue the process

(10) Repeat stepwise expansion until design completion

For each new data abstraction, repeat steps 1-9 until suitable data and program specifications are reached.

5.2. Illustrating Box Structure ADL

To illustrate how the design process works, a package will be developed to provide the following services: (1) Draw a menu on a background window, providing a controller to do tasks associated with specified function keys, and (2) create a file associated with the menu.

The first step is to load a package specification template into the editor or word processor and defining its stimuli and responses. Stimuli to a package include procedure and function names. Hence, procedure and function templates will be used during this phase.

```

package Menu
is
-----
-- Stimuli, Responses, and Transactions for Menu
--
procedure Create_File
(
-----
-- Stimuli and Responses for Create_File
--
-----
-- Transactions for Create_File
--
-----
);

procedure Controller
(
-----
-- Stimuli and Responses for Controller
--
-----
-- Transactions for Controller
--
-----
);
-----
end Menu;

```

Next specify the stimuli and responses for each of the two procedures. When declaring stimuli and responses for subprograms the need for data and data types may be discovered. Thus the definition of types and data within the package specifications (as well as package body) may continue until all black box transactions for that package have reached their clear box expansion. Also, the need to "with" other units may become apparent.

```

with Window_Manager, Direct_IO;
package Menu
is
-----
-- Stimuli, Responses, and Transactions for Menu

```

```

--
type Menu_Data is ...;

package Menu_IO is new Direct_IO(Menu_Data);

procedure Create_File
(
-----
-- Stimuli and Responses for Create_File
--
File :      out Menu_IO.File_Type
-----
-- Transactions for Create_File
--
-----
);

procedure Controller
(
-----
-- Stimuli and Responses for Controller
--
Background_Window : in      Window_Manager.Window_Type
-----
-- Transactions for Controller
--
-----
);
-----
end Menu;

```

The following notation is used in the continuing development of the black box transactions.

```

[] - a sequence of transactions
() - a condition
<> - a value

```

```

with Window_Manager, Direct_IO;
package Menu
is
-----
-- Stimuli, Responses, and Transactions for Menu
--
type Menu_Data is ...;

```

```
package Menu_IO is new Direct_IO(Menu_Data);
```

```
procedure Create_File
```

```
(  
-----  
-- Stimuli and Responses for Create_File  
--  
File :      out Menu_IO.File_Type  
-----  
-- Transactions for Create_File  
--  
-- [A]  
-----  
);
```

```
procedure Controller
```

```
(  
-----  
-- Stimuli and Responses for Controller  
--  
Background_Window : in      Window_Manager.Window_Type  
-----  
-- Transactions for Controller  
--  
-- [A]  
-- [B]  
-- case  
--   <keystroke1>  
-- is  
-- when  
--   F1-key  
-- =>  
--   null;  
-- when  
--   F2-key  
-- =>  
--   if  
--     (P)  
--   then  
--     [C]  
--   else  
--     [D]  
--   end if;  
-- when  
--   F3-key  
-- =>  
--   [E]  
--   case  
--     <Keystroke2>  
--   is  
--   when  
--     F1-key
```

```

--      =>
--      null;
--      when
--      F2-key
--      =>
--      [F]
--      when
--      F3-key
--      =>
--      [G]
--      end case;
-- end case;
-----
);
-----
end Menu;

```

Next, define the package body using body specifications for the boxes represented in the package specification. At this level state data is not needed for the package body. Each box is then examined and only *Controller* is found to need state data, a file pointer for the loop. The state transaction rules for *Controller* are the black box rules modified to reflect access to this state data. This is indicated by the prime " ' " symbol on some of the transaction rules. Notice that [A] in *Controller* has a prime while no state accesses are reflected in *Create_File* which uses the same procedural statements. This is because this process as defined in *Controller* is acting on *Controller's* state data, while in *Create_File* it is acting on a stimulus. One might anticipate that *Create_File* will become a common service as the development progresses.

```

package body Menu
is
-----

```

```

-- Clear Box Transactions for Menu
--
procedure Create_File
(
-----
-- Stimuli and Responses for Create_File
--
File :      out Menu_IO.File_Type
-----
)
is
begin
end Create_File;

procedure Controller
(
-----
-- Stimuli and Responses for Controller
--
Background_Window : in      Window_Manager.Window_Type
-----
)
is
-----
-- State for Controller
--
File_Pointer : Menu_IO.Positive_Count;
-----
-- State Box Transactions for Controller
--
-- [A']
-- [B]
-- case
--   <keystroke1>
-- is
-- when
--   F1-key
-- =>
--   null;
-- when
--   F2-key
-- =>
--   if
--     (P)
--   then
--     [C']
--   else
--     [D']
--   end if;
-- when
--   F3-key
-- =>

```

```

--      [E]
--      case
--          <Keystroke2>
--      is
--      when
--          F1-key
--      =>
--          null;
--      when
--          F2-key
--      =>
--          [F']
--      when
--          F3-key
--      =>
--          [G']
--      end case;
--  end case;
-----
begin
end Controller;
-----
end Menu;

```

Next develop the clear box for each of the two procedures. Modularize *Controller* by making some logically related transaction rules the black box rules for other procedures or functions. A calling sequence is left in the place of each such abstraction. Notice the following:

- (1) *Create_File* has been recognized as a common service. The transaction rule [A] at the front of *Controller* was observed to perform the same function as *Create_File*.
- (2) The result of an F2-key was observed to behave like a function. Thus these transaction rules became the black box to a function.
- (3) The new black boxes extracted from *Controller* were declared separate to *Menu*, thus allowing their development to proceed independently of one another.
- (4) The transaction rules associated with the F3-key were extracted as a package.

Brackets have been removed from some transactions to indicate

that these rules have been translated into a sufficient sequence of Ada statements.

```
package body Menu
is
-----
-- Clear Box Transactions for Menu
--
procedure Create_File
(
-----
-- Stimuli and Responses for Create_File
--
File :      out Menu_IO.File_Type
-----
)
is
-----
-- Clear Box Transactions for Create_File
--
begin
  A
  -----
end Create_File;

function Do_C_or_D
(
-----
-- Stimuli and Responses for Do_C_or_D
--
-----
-- Transactions for Do_C_or_D
--
-- if
--   (P)
-- then
--   [C']
-- else
--   [D']
-- end if;
-----
) return Menu_IO.Count is separate;

package F3_Package
is
-----
-- Stimuli, Responses, and Transactions for F3_Package
--
procedure Controller
(
-----
```

```

-- Stimuli and Responses for Controller
--
Background_Window : in      Window_Manager.Window_Type
-----
-- Transactions for Controller
--
-- [E]
-- case
--   <Keystroke2>
-- is
-- when
--   F1-key
-- =>
--   null;
-- when
--   F2-key
-- =>
--   [F']
-- when
--   F3-key
-- =>
--   [G']
-- end case;
-----
);
-----
end F3_Package;

package body F3_Package is separate;

procedure Controller
(
-----
-- Stimuli and Responses for Controller
--
Background_Window : in      Window_Manager.Window_Type
-----
)
is
-----
-- State for Controller
File_Pointer : Menu_IO.Positive_Count;
-----
-- State Box Transactions for Controller
--
-- [A']
-- [B]
-- case
--   <keystroke1>
-- is
-- when
--   F1-key

```

```

-- =>
--   null;
--   when
--     F2-key
--   =>
--     if
--       (P)
--     then
--       [C']
--     else
--       [D']
--     end if;
--   when
--     F3-key
--   =>
--     [E]
--     case
--       <Keystroke2>
--     is
--       when
--         F1-key
--       =>
--         null;
--       when
--         F2-key
--       =>
--         [F']
--       when
--         F3-key
--       =>
--         [G']
--     end case;
--   end case;
-----

-----
-- Clear Box Transactions for Controller
File      : Menu_IO.File_Type;
Keystroke1 : Window_Manager.Keystroke;
X         : Menu_IO.Count;
begin
  Create_File(File);
  B'
  case
    keystroke1
  is
  when
    F1-key
  =>
    null;
  when
    F2-key

```

```

=>
  X := Do_C_or_D;
when
  F3-key
=>
  F3_Package.Controller(Background_Window);
end case;
-----
end Controller;
-----
end Menu;

```

The remaining work continues much the same, by repeating the step-wise expansion of the newly abstracted black boxes until design completion. A verification of the clear box would then be in order.

5.3. ADL Diagrams

Diagrams of software systems have proven useful in systems analysis. Viewing a diagram is often more efficient than scrutinizing code. There are several motivations for diagramming a system (data flow analysis, usage analysis, etc.), and while one type of diagram may aid in one analysis it may obscure others. For example, a usage hierarchy diagram says nothing about control flow. A control flow diagram might be overly detailed for usage analysis. Three methods of diagramming ADL Box Structure systems for three distinct purposes are presented.

5.3.1. Control And Data Flow Diagram

Control and data flow diagrams are the result of expanding the nested black boxes of Figures 3-6. Figure 13 demonstrates

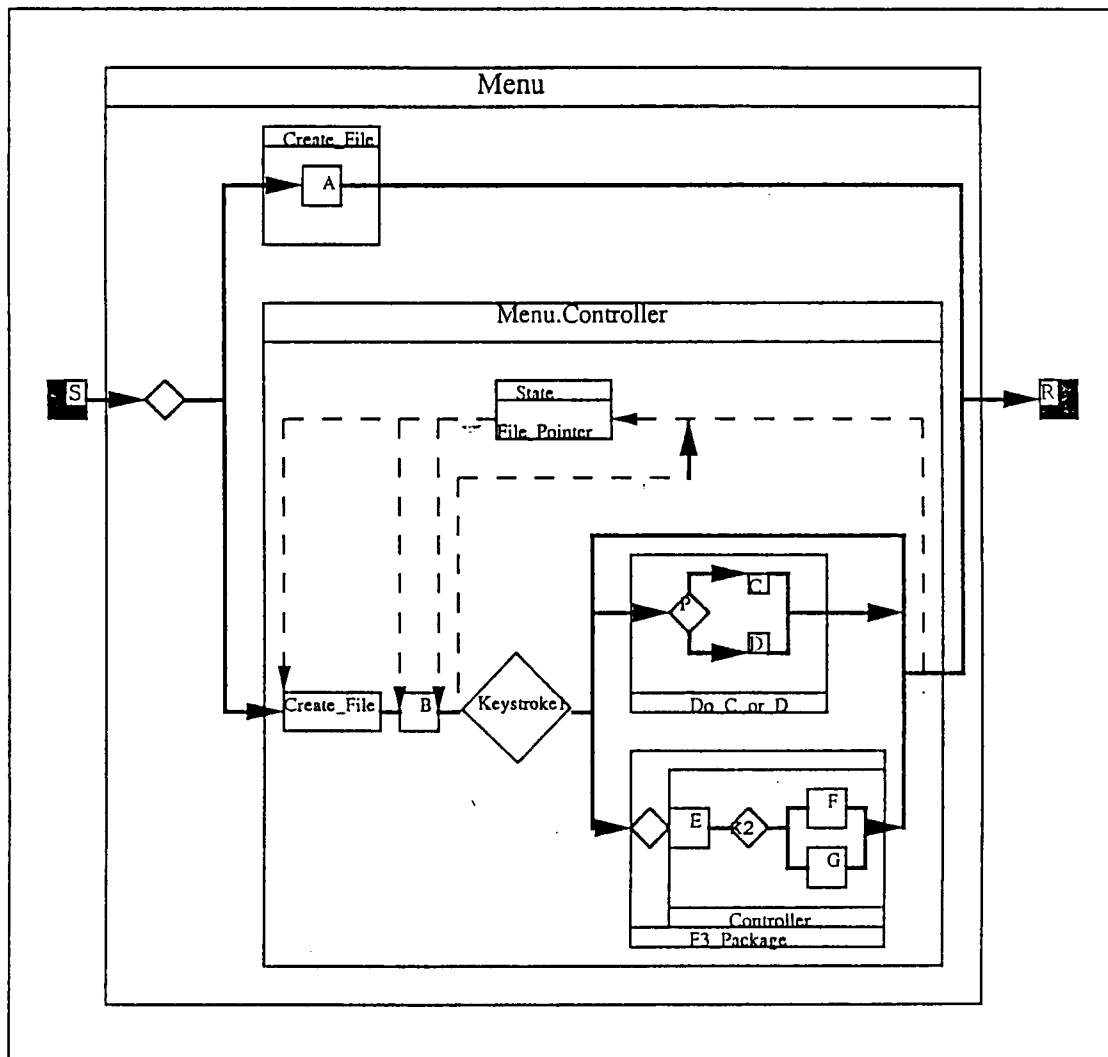


Figure 13. Control and data flow diagram of *Menu*.

such a diagram of package *Menu*. Since the package expansion was not completed, additional nested boxes could occur. Nonetheless, this diagram illustrates how the control flow paths can be represented. Data flow is illustrated by the stimulus and response lists of boxes encountered on the control flow paths.

5.3.2. Usage Vs. Availability Hierarchies

An Ada system is an interaction of subprograms. Packages serve only to provide structure for organization and information hiding of these subprograms in conjunction with their data. Hence, by ignoring packages there exists a one to one relationship between BDL usage hierarchies and ADL usage hierarchies. For example, subprogram A invokes subprograms B and D from package P. Package P contains subprograms B, C, and D - all three externally visible. Figure 14 represents the usage hierarchy of the subprograms alone.

Since a package is defined to be a black box, its effect on the hierarchy is investigated. Inserting the package into the hierarchy of Figure 14 gives rise to Figure 15. A uses P which then uses some or all of B, C, and D (in this case B and D) to provide the requested service, thus giving rise to the notion of an "availability" hierarchy rather than a usage hierarchy.

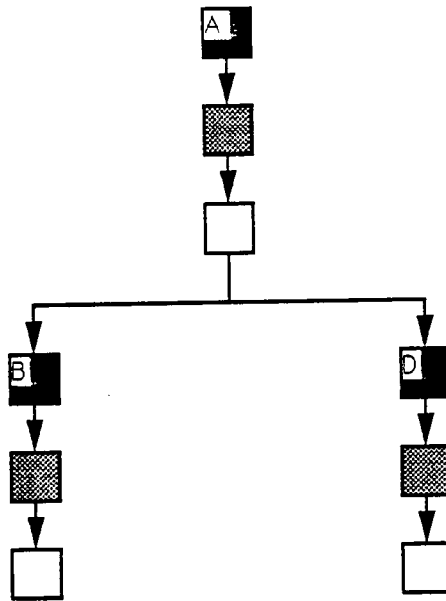


Figure 14. ADL usage hierarchy.

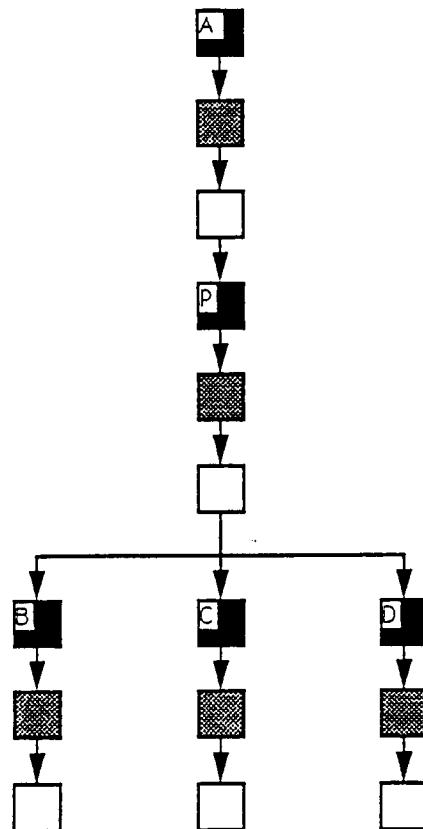


Figure 15. ADL availability hierarchy.

5.3.3. ADL Structure Diagram

During the spiral expansion process of Box Structures, such considerations as state migration require insight into the big picture of how the system is tentatively structured. The package's effect on both black boxes and data with respect to structure and visibility is paramount, as discussed in the previous section. For such analysis, usage and availability diagrams tend to withhold too much information to be of assistance. Control and data flow diagrams can be too informative and become unmanageable.

Therefore, an ADL structure diagram is needed to assist in the following analysis:

- (1) visibility,
- (2) state migration,
- (3) common services.

Two constructs are used as a basis in the representation of ADL structure diagrams. A rectangular box drawn with dashed lines will symbolize a package and a solid rectangular box will symbolize a subprogram.

There are three constructors for building an ADL structure diagram:

- (1) A subprogram can use one or more external black boxes (Figure 16).
- (2) A subprogram can use one or more internal black boxes (Figure 17).
- (3) A package can specify one or more black boxes for external usage (Figure 18).

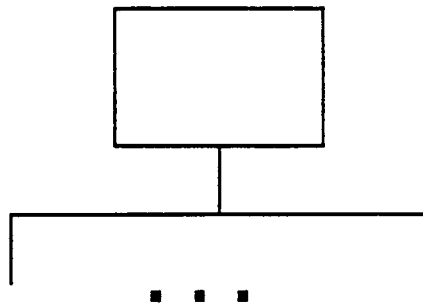


Figure 16. Subprogram invocation of external black boxes.

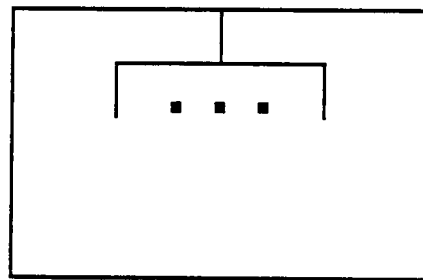


Figure 17. Subprogram invocation of internal black boxes.



Figure 18. Package specification of black boxes.

Given a non-tasking Ada system and its control and data flow diagram, the following inductive definition will construct the structure diagram for that system.

basis step:

define a structure diagram box (dashed or solid) for the outer box of the control and data flow diagram.

inductive step:

if the current structure diagram box represents a package for each black box in the packages specification place a structure diagram box on its border (according to Figure 18)

otherwise the box represents a subprogram for each black box in the control and data flow diagram being used by the current structure diagram box, attach a structure diagram box either externally (Figure 16) or internally (Figure 17) as determined by program usage.

repeat step 2 for each newly defined black box in the structure diagram.

This is illustrated by the *Menu* package. Since the initial specifications were to construct a package to provide two services, *Create_File* and *Controller*, the graph begins as Figure 19. Subsequently, two data abstractions were performed obtaining *Do_C_or_D* and the package *F3_Package* (Figure 20). The only accessible black box of *F3_Package* (*Controller*) is represented in its package specifications. *F3_Package* could now be independently developed. Meanwhile, a third black box is being invoked by *Controller*. This is the common service *Create_File*.

Observe that the first level of black boxes in a package (e.g.

Figure 19) will always represent the package specification, hence the package as a black box. Though not represented, all units are understood to contain a state (possibly null). Those units which extend from these package specification units are the consequences of data abstractions which occurred during the packages clear box expansion. Of course this inward nesting of boxes continues until all data abstractions reach their clear box expansion.

Subprogram expansions can proceed internal to that construct as well as external. Hence, the scope of a unit is being implied by the graph. When a subprogram makes use of an internal unit this denotes a locally defined unit. Externally used units are then externally visible through either a declaration or use of "with". For example, Figure 20 has *Controller* using two subprograms and a package. One

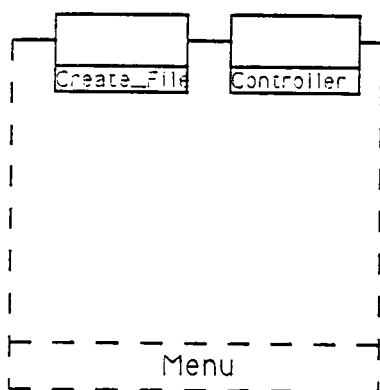


**Figure 19. Black box
for Menu.**

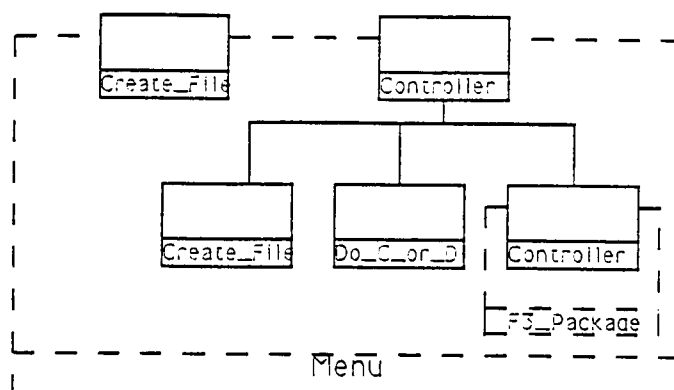


**Figure 20. Clear box
for Menu.**

subprogram, *Create_File*, is a common service and must be externally visible. However, *Do_C_or_D* as well as package *F3_Package* are only used by *Controller*. Thus, a development might enforce a stricter visibility with a structure diagram such as Figure 21. These two hierarchies are execution equivalent.

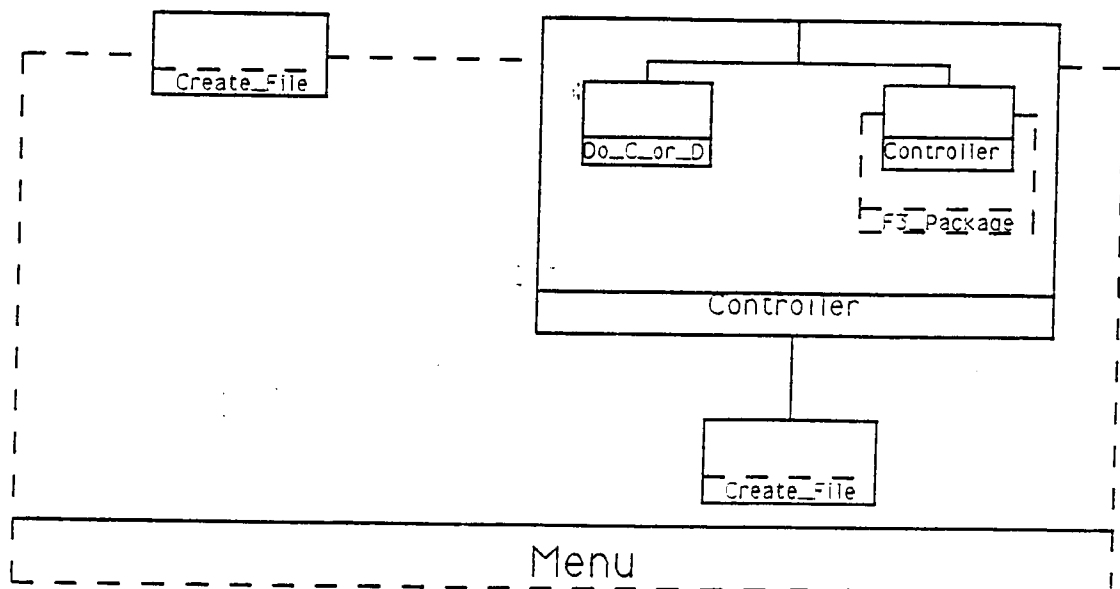


Figure 21. Clear box for *Menu*.

Chapter 6

AN ADL CASE STUDY

6.1. Introduction

A case study using this ADL was done in a DARPA-STARs project to combine the STARs environment and cleanroom development methods [10]. The cleanroom methodology uses box structures in describing a comprehensive life cycle. The STARs-cleanroom software development model [5] was applied to a small system, called the Cleanroom Reliability Manager (CRM), as a case study [6].

The appendix contains the box structure ADL package *Screen5_F6*, a third level subsystem of CRM which displays and performs the operations for an edit window. The code reflects the use of a window manager package, common services, and data types not defined in the appendix. However, the subsystem is documented and sufficiently self-contained to illustrate the box structure ADL.

6.2. Implementing An Edit Window Package

Screen5_F6 is a data abstraction resulting from the clear box expansion of the window controller for the fifth screen of CRM. The result of selecting the F6 key when on screen 5 is a pop-up edit window which redefines the "F" keys for its own

purposes. For this reason, *Screen5_F6* was abstracted as a controller for this edit window.

The black box for *Screen5_F6* is shown on pp. 71-74. The package provides one operation, the window controller. Hence, *Controller* and its associated stimuli are the only stimuli to *Screen5_F6*. The transaction rules are a mix of ADL control structures, Ada statements, and English notation.

The state box, pp. 75-79, has three pieces of state data. "Edit_Top", "Edit_Bottom", and "Edit_Light_Bar" are pointers to a file of data being manipulated by this window. Observe that the state box transaction rules have been augmented to reflect accesses to this state data.

The clear box of *Screen5_F6*, pp. 79-87, shows the clear box of *Controller* preceded by the black boxes developed during *Controller*'s clear box expansion. The subsequent pages contain the state and clear box expansions of these abstracted black boxes. By use of Ada's separate compilation these subprogram bodies were developed and declared separately from *Screen5_F6*.

Figure 22 is the ADL structure diagram for the black boxes defined in the appendix.

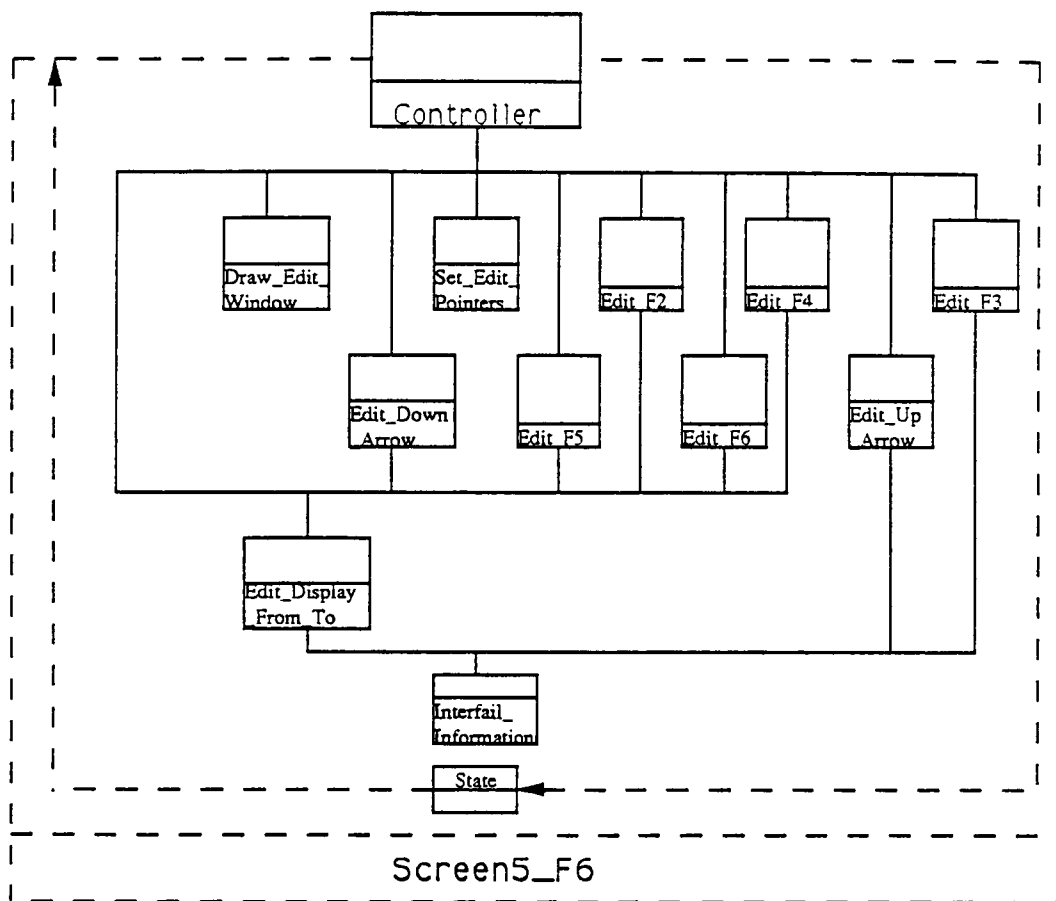


Figure 22. Structure diagram for *Screen5_F6*.

6.3. Observations

The Ada statements in *Controller*'s black box transaction rules, such as "Next_Version := Next_Version - 1;" (p. 71), reflect accesses to state data at higher levels of the usage hierarchy. These statements were defined prior to a clear box abstraction which migrated them down the usage hierarchy. In this case Next_Version is state data to the screen 5 controller. Yet when *Screen5_F6* was abstracted with transaction rules accessing Next_Version, Next_Version became a stimulus to the abstracted black box. This is important for implementing referential transparency, common services, and reusability. For this reason, all stimuli to a black box should be listed as formal arguments to the module.

Because of visibility requirements, the designer will observe a spiral effect to most ADL box structure editing sessions. For instance, while developing the clear box of *Controller*, *Edit_F2* is abstracted. One must move upward in the text editor to declare *Edit_F2* in order to reference it from within *Controller*. Meanwhile, the usage hierarchy for these abstracted black boxes proceeds downward.

Chapter 7

CONCLUSION

Box Structure ADL constructs a hierarchy of black box, state box, and clear box expansions of a system's function. The Box Structure Methodology provides the stepwise process and mathematical rigor. Ada provides the outer syntax that permits the compilation and execution of the completed system. Furthermore, the Ada language is effective for application of modern software engineering principles.

The case study referred to in the previous chapter is a thoroughly documented application of Box Structure ADL. 11855 lines of code resulted from this project of which 6740 were non-commented Ada (about 55 percent). During the year preceding this study, the writer participated in a similar case study using BDL [13]. After the systems development, the BDL was transliterated into 7200 lines of C. In this writer's opinion, the ADL was far superior to the BDL for the following reasons:

- (1) The transliteration from BDL to C cost two weeks of group project time (approximately 8 man-weeks).
- (2) The transliteration introduced many compiler and executable errors.
- (3) ADL provided an effective language for the application of information hiding, data abstraction, and modularization, unlike BDL.

The Box Structure ADL, coupled with the expansion process, created a natural and productive means for system development. Referential transparency was consistently achieved. This principle was tested by concurrently developing two of the four top level packages. The communication necessary to successfully develop these two packages was negligible.

Common services were discovered prior to redundant expansions. For instance, when a black box was viewed as a potential common service, the expansion of that black box might be delayed. Delaying its expansion allowed the gathering of information that could result in changes to the black box definition, adapting the black box to become a common service. Otherwise this common service might have been recognized only after expanding the black box with a definition too general or too specific to serve as a common service.

More research needs to be conducted on how to design systems in Box Structure ADL. BSM is an effective means for object oriented designs. More attention should be devoted to this topic. In addition, a tasking template and the associated control structures are needed to complete the conversion of BDL to ADL.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] Booch, Grady. Software Components With Ada. The Benjamin/Cummings Publishing Company, Inc. 1987.
- [2] Comer, Edward R. Ada Box Structures Methodology Handbook. Software Productivity Solutions, Inc., Melbourne, Florida, 1989.
- [3] Department of Defense, Ada Joint Program Office. Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A. Washington, D. C.: Government Printing Office, Feb. 1983.
- [4] Hevner, Alan R. "A Box-Structured Ada Design Language (ADL)." STARS Task IR-63, 1989.
- [5] Poore, J. H., David Mutchler, Harlan D. Mills. STARS-Cleanroom Reliability: Cleanroom Ideas In The STARS Environment, Software Engineering Technology, Inc., IBM P.O.308334-PJ, September, 1989.
- [6] Poore, Jesse H., Harlan D. Mills, Scott L. Hopkins, James A. Whittaker. Cleanroom Reliability Manager: A Case Study Using Cleanroom With Box Structures ADL, Software Engineering Technology, Inc., IBM P.O.308334-PJ, June, 1990.
- [7] Mills, H. D., R. C. Linger, A. R. Hevner. "Box Structured Information Systems." IBM Systems Journal V. 26, N. 4, 1987.
- [8] Mills, Harlan D., Richard C. Linger, Alan R. Hevner. Principles of Information Systems. Orlando: Academic Press, Inc., 1986.
- [9] Mills, Harlan D. "Stepwise Refinement and Verification in Box-Structured Systems." Computer June 1988, pp. 23-36.
- [10] Mills, Harlan D., Michael Dyer, Richard C. Linger. "Cleanroom Software Engineering." IEEE Transactions On Software Engineering September 1987, pp. 19-25.
- [11] Meyer, Bertrand. Object-Oriented Software Constructions. Cambridge: University Press, 1988.
- [12] Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM, vol. 5, no. 12, Dec. 1972, pp. 1053-1058.

- [13] Senn, Charles C. "Box Structure BDL Generator: A Clerical Tool to Support Software Development." Thesis, University of Tennessee, August 1989.
- [14] STARS, Software Technology for Adaptable, Reliable Systems (STARS), Statement of Work, STARS Joint Program Office, October 1987.
- [15] Wirth, Niklaus. "Program Development by Stepwise Refinement." Communications of the ACM, V.14, N.4, 1971, pp. 221-227.

APPENDIX

APPENDIX

```
package Screen5_F6
is
```

```
-----
-- Stimuli, Responses and Transactions for Screen5_F6
--
procedure Controller
(
-----
-- Stimuli and Responses for Screen5_F6
--
Current_Project      : in      Project;
Ift                  : in out Interfail_Pointer;
Next_Version         : in out Natural;
Top                  : in      Natural;
Bottom               : in      Natural;
Light_Bar_Position   : in      Natural
-----
-- Transactions for Screen5_F6
--
-- if
--   the light bar is resting on an empty line
-- then
--   display Message-5.3 in the Message Window.
-- else
--   pop the data edit window for the version on which
--   the light bar rests, set the light bar of the edit
--   window on the last line. All function keys now
--   apply to the data edit window, but the functions
--   are so similar that there is no need to rename
--   them.
--
--   loop
--     case
--       key
--     when
--       F1
--     =>
--       Record data adjustments to the Ift array.
--       Recalculate all information from this version
--       number to the end of the file. Remove the data
--       edit window. Redisplay screen 5.
--     when
--       F2
--     =>
--       if
--         there are multiple interfail times
--       then
--         delete the row on which the light bar rests.
```

```

--      Ift(edit window light bar..Next_Version-1) :=
--      Ift(edit window light bar+1..Next_Version);
--      Next_Version := Next_Version - 1;
--      else
--          -- there is only one interfail time
--          display Message-5.7
--      end if;
--  when
--      F3
--  =>
--      Append the entire Ift array (data for all
--      versions) to the .CMP file.
--      Display Message-5.8.
--  when
--      F4
--  =>
--      Display-5.9 and collect keystrokes in the
--      Message Window. Take the failure number that
--      the light bar is on and use that number in the
--      Message Window.
--      if
--          a valid interfail time is entered
--      then
--          Ift(edit window light bar+1..Next_Version+1)
--          := Ift(edit window light bar..Next_Version);
--          Ift(edit window light bar).Time :=
--              <keystrokes>;
--          Next_Version := Next_Version + 1;
--          Update the data edit window.
--      else
--          Cancel the F4 by clearing the Message Window
--          and restoring the data edit window.
--          if
--              the invalid time was numeric
--          then
--              display Message-5.6
--          else
--              display Message-5.3
--          end if;
--      end if;
--  when
--      F5
--  =>
--      Display Message-5.9 and collect keystrokes in
--      the Message Window. Use the next sequential
--      failure number beyond the last recorded for
--      this version in the Message Window.
--      if
--          a valid interfail time is entered
--      then
--          Ift(versions last failure+1..Next_Version+1)
--          := Ift(versions last failure..Next_Version);

```

```

--      Next_Version := Next_Version + 1;
--      Ift(versions last failure) := (Ift (edit
--          window light bar).Version,<keystrokes>);
--      Update the data edit window.
--  else
--      Cancel the F5 by clearing the Message Window
--      and restoring the data edit window.
--      if
--          the invalid time was numeric
--      then
--          display Message-5.6
--      else
--          display Message-5.3
--      end if;
--  end if;
--  when
--      F6
--  =>
--      Display Message-5.9 and collect keystrokes in
--      the Message Window. Take the failure number
--      that the light bar is on and use that number in
--      the Message Window.
--      if
--          a valid interfail time is entered
--      then
--          Ift(edit window light bar).Time :=
--              <keystrokes>;
--          Update the data edit window.
--      else
--          Cancel the F4 by clearing the Message Window
--          and restoring the data edit window.
--          if
--              the invalid time was numeric
--          then
--              display Message-5.6
--          else
--              display Message-5.3
--          end if;
--      end if;
--  when
--      Up-arrow-key
--  =>
--      if
--          the light bar is not on the top line of the
--          window
--      then
--          move it up one.
--      elsif
--          the light bar is on the top line of the
--          window, but the failure is not the first for
--          this version
--      then

```

```

--      scroll the file down one line.
--  else
--      -- the light bar is on the top line of the
--      -- window and the first recorded failure for
--      -- this version
--      response is null
--  end if;
--  when
--      Down-arrow-key
--  =>
--      if
--          the light bar is not on the bottom
--          information line of the window
--      then
--          move it down one.
--      elsif
--          the light bar is on the bottom information
--          line of the window, but the light bar is not
--          on the last recorded failure number for this
--          version
--      then
--          scroll the file up on line.
--      else
--          -- the light bar is on the bottom information
--          -- line of the window and on the last
--          -- recorded failure for this version
--          response is null
--      end if;
--  when
--      others
--  =>
--      Display Message-5.4
--  end case;
--  end loop;
-- end if;
-----
);
-----
end Screen5_F6;

```

```
package body Screen5_F6
is
```

```
    CEW_LENGTH          : constant Natural := 12;
    Edit_Background_Window : Window_Manager.Window_Type;
    Edit_Window          : Window_Manager.Window_Type;
-----
    -- State for Controller
    --
    Edit_Top      : Natural;
    Edit_Bottom   : Natural;
    Edit_Light_Bar : Natural;
-----
    -- State Transactions for Controller
    --
    -- if
    --   the light bar is resting on an empty line
    -- then
    --   display Message-5.3 in the Message Window.
    -- else
    --   pop the data edit window for the version on which
    --   the light bar rests, set the light bar of the edit
    --   window on the last line. All function keys now
    --   apply to the data edit window, but the functions are
    --   so similar that there is no need to rename them.
    --   Edit_Bottom := the last interfail for this version
    --   Edit_Top := Max(the first interfail for this version,
    --   Edit_Bottom-CEW_LENGTH+1);
    -- loop
    --   case
    --     key
    --   when
    --     F1
    --   =>
    --     Record data adjustments to the .CMC file.
    --     Recalculate all information from this version
    --     number to the end of the file. Remove the data
    --     edit window. Redisplay screen 5.
    --   when
    --     F2
    --   =>
    --     if
    --       there are multiple interfail times
    --     then
    --       delete the row on which the light bar rests.
    --       Ift(Edit_Light_Bar..Next_Version-1) :=
    --         Ift(Edit_Light_Bar+1..Next_Version);
    --       Next_Version := Next_Version - 1;
    --     if
    --       light bar was on the versions last interfail
    --     then
    --       Edit_Light_Bar := Edit_Light_Bar-1;
```

```

--      end if;
--      Edit_Bottom := Min(the last failure, the light
--      bar index + the number of records remaining
--      in the window beneath the light bar);
--      Edit_Top := Max(Light_Bar_Position,
--      Edit_Bottom-CEW_LENGTH+1);
--    else
--      -- there is only one interfail time
--      display Message-5.7
--    end if;
--  when
--    F3
--  =>
--    Append the entire Ift array (data for all
--    versions) to the .CMP file.
--    Display Message-5.8.
--  when
--    F4
--  =>
--    Display-5.9 and collect keystrokes in the
--    Message Window. Take the failure number that
--    the light bar is on and use that number in the
--    Message Window.
--    if
--      a valid interfail time is entered
--    then
--      Ift(Edit_Light_Bar+1..Next_Version+1) :=
--      Ift(Edit_Light_Bar..Next_Version);
--      Ift(Edit_Light_Bar).Time := <keystrokes>;
--      Next_Version := Next_Version + 1;
--      Edit_Bottom := Min(the last failure, the light
--      bar index + the number of records remaining
--      in the window beneath the light bar);
--      Edit_Top := Max(Light_Bar_Position,
--      Edit_Bottom-CEW_LENGTH+1);
--      Update the data edit window.
--    else
--      Cancel the F4 by clearing the Message Window
--      and restoring the data edit window.
--      if
--        the invalid time was numeric
--      then
--        display Message-5.6
--      else
--        display Message-5.3
--      end if;
--    end if;
--  when
--    F5
--  =>
--    Display Message-5.9 and collect keystrokes in
--    the Message Window. Use the next sequential

```

```

--      failure number beyond the last recorded for this
--      version in the Message Window.
--      if
--          a valid interfail time is entered
--      then
--          Ift(Versions last failure+1..Next_Version+1)
--              :=
--              Ift(Versions last failure..Next_Version);
--          Next_Version := Next_Version + 1;
--          Ift(Versions last failure) :=
--              (Ift(Edit_Light_Bar).Version,<keystrokes>);
--          Edit_Light_Bar := Versions last failure;
--          Edit_Bottom := Min(the last failure, the light
--              bar index + the number of records remaining
--              in the window beneath the light bar);
--          Edit_Top := Max(Light_Bar_Position,
--              Edit_Bottom-CEW_LENGTH+1);
--          Update the data edit window.
--      else
--          Cancel the F5 by clearing the Message Window
--          and restoring the data edit window.
--          if
--              the invalid time was numeric
--          then
--              display Message-5.6
--          else
--              display Message-5.3
--          end if;
--      end if;
--  when
--      F6
--  =>
--      Display Message-5.9 and collect keystrokes in
--      the Message Window. Take the failure number
--      that the light bar is on and use that number in
--      the Message Window.
--      if
--          a valid interfail time is entered
--      then
--          Ift(Edit_Light_Bar).Time := <keystrokes>;
--          Update the data edit window.
--      else
--          Cancel the F4 by clearing the Message Window
--          and restoring the data edit window.
--          if
--              the invalid time was numeric
--          then
--              display Message-5.6
--          else
--              display Message-5.3
--          end if;
--      end if;

```

```

--      when
--          Up-arrow-key
--      =>
--          if
--              the light bar is not on the top line of the
--              window
--          then
--              move it up one.
--              Edit_Light_Bar := Edit_Light_Bar - 1;
--          elsif
--              the light bar is on the top line of the
--              window, but the failure is not the first for
--              this version
--          then
--              scroll the file down one line.
--              Edit_Top := Edit_Top - 1;
--              Edit_Light_Bar := Edit_Light_Bar - 1;
--              Edit_Bottom := Edit_Bottom - 1;
--          else
--              -- the light bar is on the top line of the
--              -- window and the first recorded failure for
--              -- this version
--              response is null
--          end if;
--      when
--          Down-arrow-key
--      =>
--          if
--              the light bar is not on the bottom information
--              line of the window
--          then
--              move it down one.
--              Edit_Light_Bar := Edit_Light_Bar + 1;
--          elsif
--              the light bar is on the bottom information
--              line of the window, but the light bar is not
--              on the last recorded failure number for this
--              version
--          then
--              scroll the file up on line.
--              Edit_Top := Edit_Top + 1;
--              Edit_Light_Bar := Edit_Light_Bar + 1;
--              Edit_Bottom := Edit_Bottom + 1;
--          else
--              -- the light bar is on the bottom information
--              -- line of the window and on the last recorded
--              -- failure for this version
--              response is null
--          end if;
--      when
--          others
--      =>

```

```

--          Display Message-5.4
--      end case;
--  end loop;
-- end if;
-----

-----
-- Clear Box Transactions for Screen5_F6
--
procedure Draw_Edit_Window
(
-----
-- Stimuli and Responses for Draw_Edit_Window
--
Edit_Background_Window : in out
  Window_Manager.Window_Type;
Edit_Window            : in out
  Window_Manager.Window_Type
-----
)
is
-----
-- Clear Box Transactions for Draw_Edit_Window
--
begin
  Window_Manager.Create_Window
    ((4,28),(7,55),TRUE,0,Edit_Background_Window);
  Window_Manager.Set_Window_Cursor
    (Edit_Background_Window,(1,4));
  Window_Manager.Put(Edit_Background_Window,
    "Data for Version " & Failure_or_Version_String
    (Ift(Light_Bar_Position).Version));
  Window_Manager.Set_Window_Cursor
    (Edit_Background_Window,(3,3));
  Window_Manager.Put(Edit_Background_Window,
    "Failure      Time-to");
  Window_Manager.Set_Window_Cursor
    (Edit_Background_Window,(4,4));
  Window_Manager.Put(Edit_Background_Window,
    "Number      Failure");
  Window_Manager.Create_Window
    ((9,28),(20,55),TRUE,0,Edit_Window);
-----
end Draw_Edit_Window;

procedure Set_Edit_Pointers
(
-----
-- Stimuli and Responses for Set_Edit_Pointers
--
Ift                : in      Interfail_Pointer;
Next_Version       : in      Natural;

```

```

    Light_Bar_Position : in      Natural;
    Edit_Top           : in out Natural;
    Edit_Bottom        : in out Natural;
    Edit_Light_Bar     : in      Natural;
    Location            : in      Natural
    -----
)
is
begin
    Edit_Bottom := Min(Increment(Ift, Light_Bar_Position)-1,
        Edit_Light_Bar+CEW_LENGTH-Location);
    Edit_Top :=
        Max(Light_Bar_Position, Edit_Bottom-CEW_LENGTH+1);
    -----
end Set_Edit_Pointers;

function Interfail_Information
(
    The_Number : in      Natural;
    The_Time   : in      Interfail_Time
)
return String is
begin
    return "    " & Failure_or_Version_String(The_Number) &
        "    " & Mttf_String(The_Time);
end Interfail_Information;

procedure Edit_Display_From_To
(
    -----
    -- Stimuli and Responses for Edit_Display_From_To
    --
    Edit_Window : in      Window_Manager.Window_Type;
    Ift          : in      Interfail_Pointer;
    Top          : in      Natural;
    From         : in      Natural;
    To          : in      Natural
    -----
)
is
begin
    Window_Manager.Set_Window_Cursor
        (Edit_Window, (Window_Manager.Line(From-Top+1), 1));
    Window_Manager.Erase_In_Window
        (Edit_Window, Window_Manager.Cursor_To_End);
    for
        I in From .. To
    loop
        Window_Manager.Set_Window_Cursor(Edit_Window,
            (Window_Manager.Line(I-Top+1), 1));
        Window_Manager.Put
            (Edit_Window, Interfail_Information(I, Ift(I).Time));
    end loop;
end Edit_Display_From_To;

```

```

    end loop;
-----
end Edit_Display_From_To;

procedure Edit_F2
(
-----
  -- Stimuli and Responses for Edit_F2
  --
  Edit_Window      : in      Window_Manager.Window_Type;
  Ift               : in out Interfail_Pointer;
  Next_Version     : in out Natural;
  Edit_Top         : in out Natural;
  Edit_Bottom      : in out Natural;
  Edit_Light_Bar   : in out Natural
-----
  -- Transactions for Edit_F2
  --
  -- if
  --   there are multiple interfail times
  -- then
  --   delete the row on which the light bar rests.
  --   Ift(Edit_Light_Bar..Next_Version-1) :=
  --     Ift(Edit_Light_Bar+1..Next_Version);
  --   Next_Version := Next_Version - 1;
  --   if
  --     light bar was on the versions last interfail
  --   then
  --     Edit_Light_Bar := Edit_Light_Bar-1;
  --   end if;
  --   Edit_Bottom := Min(the last failure, the light
  --     bar index + the number of records remaining in
  --     the window beneath the light bar);
  --   Edit_Top := Max(Light_Bar_Position,
  --     Edit_Bottom-CEW_LENGTH+1);
  -- else
  --   -- there is only one interfail time
  --   display Message-5.7
  -- end if;
-----
) is separate;

```

```

procedure Edit_F3
(
-----
  -- Stimuli and Responses for Edit_F3
  --
  Current_Project  : in      Project;
  Ift               : in      Interfail_Pointer;
  Next_Version     : in      Natural
-----
  -- Transactions for Edit_F3

```

```

--
-- Append the entire Ift array (data for all versions)
-- to the .CMP file.
-- Display Message-5.8.
-----
) is separate;

procedure Edit_F4
(
-----
-- Stimuli and Responses for Edit_F4
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift               : in out Interfail_Pointer;
Next_Version     : in out Natural;
Edit_Top         : in out Natural;
Edit_Bottom      : in out Natural;
Edit_Light_Bar   : in out Natural
-----

-- Transactions for Edit_F4
--
-- Display-5.9 and collect keystrokes in the
-- Message Window. Take the failure number that
-- the light bar is on and use that number in the
-- Message Window.
-- if
--   a valid interfail time is entered
-- then
--   Ift(Edit_Light_Bar+1..Next_Version+1) :=
--     Ift(Edit_Light_Bar..Next_Version);
--   Ift(Edit_Light_Bar).Time := <keystrokes>;
--   Next_Version := Next_Version + 1;
--   Edit_Bottom := Min(the last failure, the light
--     bar index + the number of records remaining in
--     the window beneath the light bar);
--   Edit_Top := Max(Light_Bar_Position,
--     Edit_Bottom-CEW_LENGTH+1);
--   Update the data edit window.
-- else
--   Cancel the F4 by clearing the Message Window
--   and restoring the data edit window.
--   if
--     the invalid time was numeric
--   then
--     display Message-5.6
--   else
--     display Message-5.3
--   end if;
-- end if;
-----
) is separate;

```

```
procedure Edit_F5
```

```
(
```

```
-----  
-- Stimuli and Responses for Edit_F5
```

```
--  
Edit_Window      : in      Window_Manager.Window_Type;  
Ift              : in out Interfail_Pointer;  
Next_Version     : in out Natural;  
Top              : in      Natural;  
Light_Bar_Position : in      Natural;  
Edit_Top         : in out Natural;  
Edit_Bottom      : in out Natural;  
Edit_Light_Bar   : in out Natural  
-----
```

```
-- Transactions for Edit_F5
```

```
--  
-- Display Message-5.9 and collect keystrokes in  
-- the Message Window. Use the next sequential  
-- failure number beyond the last recorded for this  
-- version in the Message Window.  
-- if  
--   a valid interfail time is entered  
-- then  
--   Ift(Versions last failure+1..Next_Version+1)  
--   :=  
--   Ift(Versions last failure..Next_Version);  
--   Next_Version := Next_Version + 1;  
--   Ift(Versions last failure) :=  
--   (Ift(Edit_Light_Bar).Version,<keystrokes>);  
--   Edit_Light_Bar := Versions last failure;  
--   Edit_Bottom := Min(the last failure, the light  
--   bar index + the number of records remaining in  
--   the window beneath the light bar);  
--   Edit_Top := Max(Light_Bar_Position,  
--   Edit_Bottom-CEW_LENGTH+1);  
--   Update the data edit window.  
-- else  
--   Cancel the F5 by clearing the Message Window  
--   and restoring the data edit window.  
--   if  
--     the invalid time was numeric  
--   then  
--     display Message-5.6  
--   else  
--     display Message-5.3  
--   end if;  
-- end if;
```

```
-----  
) is separate;
```

```
procedure Edit_F6
```

```
(
```

```

-----
-- Stimuli and Responses for Edit_F6
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in out  Interfail_Pointer;
Edit_Top         : in      Natural;
Edit_Bottom      : in      Natural;
Edit_Light_Bar   : in      Natural
-----

```

```

-----
-- Transactions for Edit_F6
--
-- Display Message-5.9 and collect keystrokes in
-- the Message Window. Take the failure number
-- that the light bar is on and use that number in
-- the Message Window.
-- if
--   a valid interfail time is entered
-- then
--   Ift(Edit_Light_Bar).Time := <keystrokes>;
--   Update the data edit window.
-- else
--   Cancel the F4 by clearing the Message Window
--   and restoring the data edit window.
--   if
--     the invalid time was numeric
--   then
--     display Message-5.6
--   else
--     display Message-5.3
--   end if;
-- end if;
-----

```

) is separate;

procedure Edit_Up_Arrow

```

(
-----
-- Stimuli and Responses for Edit_Up_Arrow
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in      Interfail_Pointer;
Top              : in      Natural;
Light_Bar_Position : in      Natural;
Edit_Top         : in out  Natural;
Edit_Bottom      : in out  Natural;
Edit_Light_Bar   : in out  Natural
-----
-- Transactions for Edit_Up_Arrow
--
-- if
--   the light bar is not on the top line of the
--   window

```

```

-- then
--   move it up one.
--   Edit_Light_Bar := Edit_Light_Bar - 1;
-- elsif
--   the light bar is on the top line of the
--   window, but the failure is not the first for
--   this version
-- then
--   scroll the file down one line.
--   Edit_Top := Edit_Top - 1;
--   Edit_Light_Bar := Edit_Light_Bar - 1;
--   Edit_Bottom := Edit_Bottom - 1;
-- else
--   -- the light bar is on the top line of the
--   -- window and the first recorded failure for
--   -- this version
--   response is null
-- end if;
-----

```

) is separate;

procedure Edit_Down_Arrow

```

(
-----
-- Stimuli and Responses for Edit_Down_Arrow
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in      Interfail_Pointer;
Top              : in      Natural;
Light_Bar_Position : in      Natural;
Edit_Top         : in out Natural;
Edit_Bottom      : in out Natural;
Edit_Light_Bar   : in out Natural
-----
-- Transactions for Edit_Down_Arrow
--
-- if
--   the light bar is not on the bottom information
--   line of the window
-- then
--   move it down one.
--   Edit_Light_Bar := Edit_Light_Bar + 1;
-- elsif
--   the light bar is on the bottom information
--   line of the window, but the light bar is not
--   on the last recorded failure number for this
--   version
-- then
--   scroll the file up on line.
--   Edit_Top := Edit_Top + 1;
--   Edit_Light_Bar := Edit_Light_Bar + 1;
--   Edit_Bottom := Edit_Bottom + 1;

```

```

-- else
--   -- the light bar is on the bottom information
--   -- line of the window and on the last recorded
--   -- failure for this version
--   response is null
-- end if;
-----
) is separate;

procedure Controller
(
-----
-- Stimuli and Responses for Screen5_F6
--
Current_Project      : in      Project;
Ift                  : in out Interfail_Pointer;
Next_Version         : in out Natural;
Top                  : in      Natural;
Bottom              : in      Natural;
Light_Bar_Position   : in      Natural
-----
)
is
-----
-- Clear Box Transactions for Controller
Key : Window_Manager.Keystroke;
begin
  Draw_Edit_Window(Edit_Background_Window,Edit_Window);
  Edit_Light_Bar := Increment(Ift,Light_Bar_Position)-1;
  Set_Edit_Pointers(Ift,Next_Version,Light_Bar_Position,
    Edit_Top,Edit_Bottom,Edit_Light_Bar,CEW_LENGTH);
  Edit_Display_From_To(Edit_Window,Ift,Edit_Top,
    From=>Edit_Top,To=>Edit_Bottom);
  Window_Services.Toggle_Light_Bar(Edit_Window,
    Window_Services.ON,Positive(Edit_Light_Bar-
    Edit_Top+1));
  loop
    if
      Next_Version = Ift'last
    then
      Rebuild_Ift(Ift,Next_Version);
    end if;
    Key := Window_Manager.Get_Keystroke;
    if
      Key.The_Set = Window_Manager.Labelled
    then
      case
        Key.The_Key
      is
        when
          Window_Manager.F_1_Key
        =>

```

```

        exit;
    when
        Window_Manager.F_2_Key
    =>
        Edit_F2(Edit_Window,Ift,Next_Version,Edit_Top,
            Edit_Bottom,Edit_Light_Bar);
    when
        Window_Manager.F_3_Key
    =>
        Edit_F3(Current_Project,Ift,Next_Version);
    when
        Window_Manager.F_4_Key
    =>
        Edit_F4(Edit_Window,Ift,Next_Version,Edit_Top,
            Edit_Bottom,Edit_Light_Bar);
    when
        Window_Manager.F_5_Key
    =>
        Edit_F5(Edit_Window,Ift,Next_Version,Top,
            Light_Bar_Position,Edit_Top,Edit_Bottom,
            Edit_Light_Bar);
    when
        Window_Manager.F_6_Key
    =>
        Edit_F6(Edit_Window,Ift,Edit_Top,Edit_Bottom,
            Edit_Light_Bar);
    when
        Window_Manager.Up_Arrow_Key
    =>
        Edit_Up_Arrow(Edit_Window,Ift,Top,
            Light_Bar_Position, Edit_Top,Edit_Bottom,
            Edit_Light_Bar);
    when
        Window_Manager.Down_Arrow_Key
    =>
        Edit_Down_Arrow(Edit_Window,Ift,Top,
            Light_Bar_Position, Edit_Top,Edit_Bottom,
            Edit_Light_Bar);
    when
        others
    =>
        Window_Services.Display_Message
            (MESSAGE5_4,CMW_LOCATION);
    end case;
else
    Window_Services.Display_Message
        (MESSAGE5_4,CMW_LOCATION);
end if;
Window_Manager.Set_Window_Cursor(Edit_Window,
    Window_Manager.Cursor_Location(Edit_Window));
end loop;
Window_Manager.Delete_Window(Edit_Window);

```

```
Window_Manager.Delete_Window(Edit_Background_Window);  
-----  
end Controller;  
-----  
end Screen5_F6;
```

```

procedure Edit_F2
(
-----
-- Stimuli and Responses for Edit_F2
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in out Interfail_Pointer;
Next_Version     : in out Natural;
Edit_Top        : in out Natural;
Edit_Bottom     : in out Natural;
Edit_Light_Bar  : in out Natural
-----
)
is
-----
-- Clear Box Transactions for Edit_F2
--
begin
  if
    Edit_Bottom-Edit_Top >= 1
  then
    for
      I in Edit_Light_Bar..Next_Version-1
    loop
      Ift(I) := Ift(I+1);
    end loop;
    Next_Version := Next_Version - 1;
    if
      Ift(Edit_Light_Bar).Version /= Ift(Edit_Top).Version
    then
      Edit_Light_Bar := Edit_Light_Bar-1;
    end if;
    Set_Edit_Pointers(Ift,Next_Version,Light_Bar_Position,
      Edit_Top,Edit_Bottom,Edit_Light_Bar,Edit_Light_Bar -
      Edit_Top+1);
    Edit_Display_From_To(Edit_Window,Ift,Edit_Top,
      From=>Edit_Top,To=>Edit_Bottom);
    Window_Services.Toggle_Light_Bar(Edit_Window,
      Window_Services.ON,Positive(Edit_Light_Bar-
      Edit_Top+1));
    else
      Window_Services.Display_Message
        (MESSAGE5_7,CMW_LOCATION);      end if;
    -----
end Edit_F2;

```

```

procedure Edit_F3
(
-----
-- Stimuli and Responses for Edit_F3
--
Current_Project : in      Project;
Ift              : in      Interfail_Pointer;
Next_Version     : in      Natural
-----
)
is
-----
-- Clear Box Transactions for Edit_F3
--
CMP : Text_IO.File_Type;
begin
  Open_CMP(CMP,Current_Project);
  Text_IO.New_Line(CMP);
  Text_IO.Put_Line(CMP,
    "          CERTIFICATION MODEL");
  Text_IO.Put_Line(CMP,
    "          Failure Data");
  Text_IO.New_Line(CMP);
  Text_IO.Put_Line
    (CMP,"Version          Failure          Time");
  Text_IO.Put_Line
    (CMP,"-----");
  for
    I in 1..Next_Version - 1
  loop
    Text_IO.Put_Line
      (CMP," " & Failure_or_Version_String(Ift(I).Version) &
        "          " & Interfail_Information(I,Ift(I).Time));
  end loop;
  Text_IO.Close(CMP);
  MESSAGE5_8.Text(1..6) :=
    Failure_or_Version_String(Next_Version-1);
  Window_Services.Display_Message(MESSAGE5_8,CMW_LOCATION);
  -----
end Edit_F3;

```

```

procedure Edit_F4
(
-----
-- Stimuli and Responses for Edit_F4
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in out Interfail_Pointer;
Next_Version     : in out Natural;
Edit_Top        : in out Natural;
Edit_Bottom     : in out Natural;
Edit_Light_Bar  : in out Natural
-----
)
is
-----
-- Clear Box Transactions for Edit_F4
--
The_Time : Interfail_Time;
Response : String(1..25);
Last     : Positive;
begin
  PROMPT5_9(42..47) :=
    Failure_or_Version_String(Edit_Light_Bar);
  Window_Services.Display_Prompt
    (PROMPT5_9, CMW_LOCATION, Response);
  Time_IO.Get(Response, The_Time, Last);
  if
    The_Time > 0.0
  then
    for
      I in reverse Edit_Light_Bar..Next_Version
    loop
      Ift(I+1) := Ift(I);
    end loop;
    Ift(Edit_Light_Bar).Time := The_Time;
    Next_Version := Next_Version + 1;
    Set_Edit_Pointers(Ift, Next_Version, Light_Bar_Position,
      Edit_Top, Edit_Bottom, Edit_Light_Bar, Edit_Light_Bar -
      Edit_Top+1);
    Edit_Display_From_To(Edit_Window, Ift, Edit_Top,
      From=>Edit_Light_Bar, To=>Edit_Bottom);
    Window_Services.Toggle_Light_Bar(Edit_Window,
      Window_Services.ON, Positive(Edit_Light_Bar-
      Edit_Top+1));
  else
    Window_Services.Display_Message
      (MESSAGE5_6, CMW_LOCATION);
  end if;
exception
  when
    Text_IO.DATA_ERROR | Text_IO.END_ERROR
  =>

```

```
Window_Services.Display_Message(MESSAGE5_4,CMW_LOCATION);  
-----  
end Edit_F4;
```

```

procedure Edit_F5
(
-----
-- Stimuli and Responses for Edit_F5
Edit_Window      : in      Window_Manager.Window_Type;
Ift               : in out Interfail_Pointer;
Next_Version     : in out Natural;
Top              : in      Natural;
Light_Bar_Position : in      Natural;
Edit_Top         : in out Natural;
Edit_Bottom      : in out Natural;
Edit_Light_Bar   : in out Natural
-----
)
is
-----
-- Clear Box Transactions for Edit_F5
The_Time        : Interfail_Time;
Response        : String(1..52);
Insert_Index    : Natural;
Last            : Positive;
begin
  Insert_Index := Increment(Ift,Light_Bar_Position);
  PROMPT5_9(42..47) :=
    Failure_or_Version_String(Insert_Index);
  Window_Services.Display_Prompt
    (PROMPT5_9,CMW_LOCATION,Response);
  Time_IO.Get(Response,The_Time,Last);
  if
    The_Time > 0.0
  then
    for
      I in reverse Insert_Index..Next_Version
    loop
      Ift(I+1) := Ift(I);
    end loop;
    Next_Version := Next_Version + 1;
    Ift(Insert_Index) :=
      (Ift(Edit_Light_Bar).Version,The_Time);
    Edit_Light_Bar := Insert_Index;
    Set_Edit_Pointers(Ift,Next_Version,Light_Bar_Position,
      Edit_Top,Edit_Bottom,Edit_Light_Bar,CEW_LENGTH);
    Edit_Display_From_To(Edit_Window,Ift,Edit_Top,
      From=>Edit_Top,To=>Edit_Bottom);
    Window_Services.Toggle_Light_Bar
      (Edit_Window,Window_Services.ON,Edit_Light_Bar-
        Edit_Top+1);
    else
      Window_Services.Display_Message
        (MESSAGE5_6,CMW_LOCATION);
    end if;
  exception

```

```

when
    Text_IO.DATA_ERROR | Text_IO.END_ERROR
=>
    Window_Services.Display_Message
        (MESSAGE5_4,CMW_LOCATION);
-----
end Edit_F5;

procedure Edit_F6
(
    -----
    -- Stimuli and Responses for Edit_F6
    Edit_Window      : in      Window_Manager.Window_Type;
    Ift              : in out Interfail_Pointer;
    Edit_Top         : in      Natural;
    Edit_Bottom      : in      Natural;
    Edit_Light_Bar   : in      Natural
    -----
)
is
    -----
    -- Clear Box Transactions for Edit_F6
    The_Time : Interfail_Time;
    Response  : String(1..25);
    Last      : Positive;
begin
    PROMPT5_9(42..47) :=
        Failure_or_Version_String(Edit_Light_Bar);
    Window_Services.Display_Prompt
        (PROMPT5_9,CMW_LOCATION,Response);
    Time_IO.Get(Response,The_Time,Last);
    if
        The_Time > 0.0
    then
        Ift(Edit_Light_Bar).Time := The_Time;
        Edit_Display_From_To(Edit_Window,Ift>Edit_Top,
            From=>Edit_Light_Bar,To=>Edit_Bottom);
        Window_Services.Toggle_Light_Bar(Edit_Window,
            Window_Services.ON,Positive(Edit_Light_Bar-
                Edit_Top+1));
    else
        Window_Services.Display_Message
            (MESSAGE5_6,CMW_LOCATION);
    end if;
exception
    when
        Text_IO.DATA_ERROR | Text_IO.END_ERROR
        =>
            Window_Services.Display_Message
                (MESSAGE5_4,CMW_LOCATION);
            -----
end Edit_F6;

```

procedure Edit_Up_Arrow

(

-- Stimuli and Responses for Edit_Up_Arrow

--

Edit_Window : in Window_Manager.Window_Type;
Ift : in Interfail_Pointer;
Top : in Natural;
Light_Bar_Position : in Natural;
Edit_Top : in out Natural;
Edit_Bottom : in out Natural;
Edit_Light_Bar : in out Natural

)

is

-- Clear Box Transactions for Edit_Up_Arrow

--

Light_Bar_Location : Positive;

begin

Light_Bar_Location := Positive(Edit_Light_Bar-Edit_Top+1);

if

Light_Bar_Location > 1

then

Window_Services.Toggle_Light_Bar(Edit_Window,
Window_Services.OFF,Light_Bar_Location);

Edit_Light_Bar := Edit_Light_Bar - 1;

Window_Services.Toggle_Light_Bar(Edit_Window,
Window_Services.ON,Light_Bar_Location-1);

elsif

Edit_Top > Light_Bar_Position

then

Window_Services.Toggle_Light_Bar
(Edit_Window,Window_Services.OFF,1);

Window_Manager.Scroll(Edit_Window,Window_Manager.Down);

Edit_Top := Edit_Top - 1;

Edit_Light_Bar := Edit_Light_Bar - 1;

Edit_Bottom := Edit_Bottom - 1;

Window_Manager.Set_Window_Cursor(Edit_Window,(1,1));

Window_Manager.Put(Edit_Window,Interfail_Information
(Edit_Light_Bar,Ift(Edit_Light_Bar).Time));

Window_Services.Toggle_Light_Bar
(Edit_Window,Window_Services.ON,1);

end if;

end Edit_Up_Arrow;

```

procedure Edit_Down_Arrow
(
-----
-- Stimuli and Responses for Edit_Down_Arrow
--
Edit_Window      : in      Window_Manager.Window_Type;
Ift              : in      Interfail_Pointer;
Top              : in      Natural;
Light_Bar_Position : in      Natural;
Edit_Top         : in out Natural;
Edit_Bottom      : in out Natural;
Edit_Light_Bar   : in out Natural
-----
)
is
-----
-- Clear Box Transactions for Edit_Down_Arrow
--
Light_Bar_Location : Positive;
begin
Light_Bar_Location := Positive(Edit_Light_Bar-Edit_Top+1);
if
    Light_Bar_Location < CEW_LENGTH and
    Edit_Light_Bar < Increment(Ift,Light_Bar_Position)-1
then
    Window_Services.Toggle_Light_Bar(Edit_Window,
        Window_Services.OFF,Light_Bar_Location);
    Edit_Light_Bar := Edit_Light_Bar + 1;
    Window_Services.Toggle_Light_Bar(Edit_Window,
        Window_Services.ON,Light_Bar_Location+1);
elsif
    Edit_Bottom < Increment(Ift,Light_Bar_Position)-1
then
    Window_Services.Toggle_Light_Bar(Edit_Window,
        Window_Services.OFF,Light_Bar_Location);
    Window_Manager.Scroll(Edit_Window,Window_Manager.Up);
    Edit_Top := Edit_Top + 1;
    Edit_Light_Bar := Edit_Light_Bar + 1;
    Edit_Bottom := Edit_Bottom + 1;
    Edit_Display_From_To(Edit_Window,Ift,Edit_Top,
        From=>Edit_Top,To=>Edit_Bottom);
    Window_Services.Toggle_Light_Bar(Edit_Window,
        Window_Services.ON,Light_Bar_Location);
end if;
-----
end Edit_Down_Arrow;

```

VITA

Scott Lee Hopkins was born in Tupelo, Mississippi on April 26, 1962. After graduating from Mooreville High School in 1980, Scott purchased a community grocery store. He managed this store for two years after which he went to Athens, Greece to work as a missionary for approximately one year.

In 1984 Scott entered the computer science program at Itawamba Junior College in Fulton, Mississippi. After two years he married Angela Kendrick and continued his education at the University of Mississippi in Oxford. In 1988 he obtained a Bachelor of Science degree in Computer Science.

Scott entered the computer science program at the University of Tennessee, Knoxville as a graduate teaching assistant in August, 1988. He received the Master of Science degree in Computer Science in August, 1990. Scott is currently employed by Martin Marietta Energy Systems, Oak Ridge, Tennessee.