

AD-A244 182



1

DTIC
ELECTE
JAN 07 1992
S D D



This document has been approved
for public release and sale; its
distribution is unlimited.

92-00166

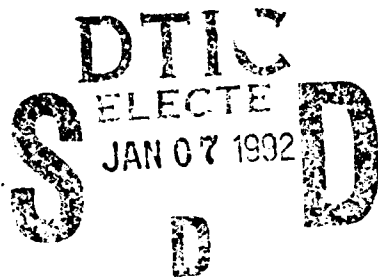


DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

00 1 0 129

AFIT/GCS/ENG/91D-6



FORMALIZATION AND VALIDATION OF
AN SADT SPECIFICATION THROUGH
EXECUTABLE SIMULATION IN VHDL

THESIS

Daniel Lee Eickmeier
Captain, USAF

AFIT/GCS/ENG/91D-6

Approved for public release; distribution unlimited

AFIT/GCS/ENG/91D-6

FORMALIZATION AND VALIDATION OF AN SADT
SPECIFICATION THROUGH EXECUTABLE
SIMULATION IN VHDL

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Daniel Lee Eickmeier, MBA, BSEE
Captain, USAF

December, 1991

ADDITIONAL INFORMATION

ADDITION FOR	
NTIS ORIGIN	J
DTIC FILE	
Unannounced	
J. Shuman	
By	
Detachable	
Available	
Dist	
A-1	

Approved for public release; distribution unlimited

Preface

This research was part of a continuing effort to establish the use of sound engineering disciplines in the field of software engineering. One such discipline is the formalization and validation of desired product behavior early in the product's development life cycle. This thesis investigates the conversion of an informal SADT specification into a formal, executable specification in the VHSIC Hardware Description Language (VHDL). I believe some of the same benefits realized in the hardware industry by using VHDL to specify integrated circuit behavior can be realized in the software industry through the use of VHDL in the specification of software system behavior. My hope is that the process described in this thesis for converting SADT specifications into executable VHDL specifications provides a starting point for realizing the benefits of this discipline.

I want to thank several people for their help in the completion of this thesis. First, I want to thank my thesis advisor, Major Kim Kanzaki, for his guidance and technical advice during this research. Additionally, I would like to thank the members of my research committee, Major Paul Bailor and Dr. Thomas Hartrum, for sharing their excitement in the advance of software engineering practices. I also want to thank Captain Randy Douglass for helping me keep things in perspective as we tackled common problems. Those 2 a.m. motorcycle excursions helped overcome many frustrating research moments.

The completion of this thesis has been a family affair. I wish to thank my loving wife, Ann, for her unending support and caring. Thanks also to my 4 year old son, Luke, and my 2 year old son, Joshua, for their patience and encouragement. They too have been diligently working on their "thesis" and going to "school" just like daddy. Finally, thanks to our God for the strength He has provided over the last 18 months, and for the safe and healthy delivery of our third son, Isaac Joel, on 29 Nov 1991, just two days after this thesis was approved!

Daniel Lee Eickmeier

Table of Contents

	Page
Preface	ii
Table of Contents	iii
List of Figures	x
List of Tables	xiii
Abstract	xvi
I. Introduction	1
1.1 Background	1
1.2 Problem	4
1.3 Research Objectives	5
1.4 Definitions	5
1.4.1 SADT	5
1.4.2 IDEF ₀	6
1.4.3 VHDL	12
1.4.4 Refine	13
1.5 Scope	14
1.6 Standards	14
1.7 Equipment and Software	16
1.8 Sequence of Presentation	16
II. Literature Review	17
2.1 Introduction	17
2.2 Prototyping Tools	17

	Page
2.2.1 PAISLey	18
2.2.2 STATEMATE	22
2.2.3 The Animator	27
2.2.4 Refine	29
2.3 Analysis Tools	30
2.3.1 PSL/PSA	30
2.4 Summary	32
III. Requirements Analysis of an SADT to VHDL Transformation Algorithm .	33
3.1 Introduction	33
3.2 Definition of the SADT Method's Primary Features	33
3.2.1 SADT Building Block Components	33
3.2.2 SADT Diagrams and Models	34
3.2.3 Special Diagram Notations for Arrows	38
3.3 Definition of VHDL's Primary Attributes	40
3.3.1 VHDL Building Block Components	40
3.3.2 VHDL Blocks and Models	41
3.4 Mapping SADT to VHDL	41
3.5 Extensions to the SADT Model	43
3.5.1 SADT Activity Activation Rules	43
3.5.2 Decision Tables	43
3.5.3 Creating and Implementing the Decision Tables	47
3.6 Levels of Executability of the VHDL Model	47
3.7 Summary	47
IV. Manual Implementation of the SADT to VHDL Transformation Algorithm	52
4.1 Introduction	52
4.2 The Heating System Problem	53

	Page
4.2.1 SADT Analysis	53
4.2.2 Problems Encountered	58
4.2.3 Decision Tables for the Leaf Node Activities	60
4.2.4 Test Cases	60
4.3 VHDL Source Code Generation for the Heating System	62
4.3.1 Types Definition Package	62
4.3.2 Entity Declarations	64
4.3.3 Architecture Body Declarations - Behavioral	65
4.3.4 Architecture Body Declarations - Structural	68
4.4 Extending the Heating System to Address Timing	74
4.5 Generating the VHDL Test Environment	78
4.5.1 System Testbench	78
4.5.2 System Structural Declaration	78
4.5.3 Component Configuration Declaration	80
4.6 Executing the VHDL Simulation of the Heating System	80
4.7 The Lift Control System Problem	82
4.7.1 SADT Analysis	83
4.7.2 Problems Encountered	92
4.7.3 Test Cases Generated	92
4.8 Code Generation and Simulation of the Lift Control System	95
4.8.1 Pipeline Arrows	95
4.8.2 Feedback Loops	98
4.8.3 Concurrent Processing of SADT Activities	100
4.8.4 Real-Time Extensions to the Lift Control System	102
4.8.5 Executing the VHDL Simulation of the Lift Control System	103
4.9 Steps Taken in Transforming SADT into VHDL Source Code	105
4.10 Summary	107

	Page
V. Requirements Analysis and Design of an SADT to VHDL Transformer . .	109
5.1 Introduction	109
5.2 The Existing Model	109
5.2.1 Addition Needed to the Abstract Model	109
5.3 Requirements for the SADT Transformer	110
5.4 Design of the AM-Reader Object	113
5.4.1 Reading in an Existing Project	113
5.4.2 Processing the Data Elements	113
5.4.3 Processing the Activities	115
5.4.4 Generic Relationship Manager	120
5.5 Generation of the VHDL Source Code	121
5.5.1 Completeness of the Code Generation	125
5.5.2 Cautions of the Automated Process	126
5.6 Summary	127
VI. Comparison of the VHDL and Refine Implementation Methods	128
6.1 Developing the SADT's	128
6.1.1 Implied Sequential Ordering	129
6.1.2 Feedback Loops	129
6.2 Creating the Decision Tables	132
6.2.1 Mutual Exclusion	133
6.2.2 Completeness	136
6.2.3 Combinatoric Explosion	136
6.3 Performing the Translation	137
6.3.1 Determining the Mappings	137
6.3.2 Generating the Source Code	138
6.4 Running the Simulations	140
6.4.1 Generating the Test Cases	140

	Page
6.4.2 Compiling the Source Code	141
6.4.3 Execution of the Simulation	143
6.5 Summary	144
VII. Conclusions and Recommendations	146
7.1 Introduction	146
7.2 Summary of Accomplishments	146
7.3 Conclusions	147
7.4 Recommendations for Additional Research	149
7.4.1 Automation of the SADT to VHDL Translation	149
7.4.2 Additional Methods for Specifying Behavior	149
Appendix A. The Heating System Problem	152
A.1 Decision Tables	152
Appendix B. The Lift Control System Problem	161
B.1 Decision Tables	161
Appendix C. The Heating System Source Code	197
C.1 Type Definitions Package	197
C.2 Architecture Structure of System	198
C.3 Entity Declaration for Control_Heater (A-0)	201
C.4 Architecture Structure of Control_Heater (A-0)	202
C.5 Entity Declarations for the A0 Level	204
C.6 Architecture Structures for the A0 Level	206
C.7 Architecture Behavior for the A0 Level	209
C.8 Entity Declarations for the A2 Level	210
C.9 Architecture Behavior for the A2 Level	211
C.10 Entity Declarations for the A3 Level	213

	Page
C.11 Architecture Behavior for the A3 Level	214
C.12 Entity Declarations for Timers	216
C.13 Architecture Behavior for Timers	217
C.14 Entity and Architecture for the Testbench	218
C.15 Heater System Configuration File	231
C.16 Heater System Compilation Script	233
C.17 Heater System Simulation Script	234
C.18 Heater System Simulation Include File	235
C.19 Heater System Simulation Output	237
Appendix D. The Lift Control System Problem Source Code	243
Appendix E. Lift Control System Simulation Output	244
E.1 Lift Control System Testbench Entity and Architecture Declarations	244
E.2 Lift Control System Simulation Output	257
Appendix F. Objects and Operations of the Essential Data Model for IDEF ₀ Diagrams	276
F.1 ACTIVITY-CLASS	276
F.1.1 Record Fields	276
F.1.2 Operations	276
F.2 DATA-ELEMENT-CLASS	277
F.2.1 Record Fields	277
F.2.2 Operations	278
F.3 ICOM-RELATION-CLASS	278
F.3.1 Record Fields	278
F.3.2 Operations	279
F.4 CALLS-RELATION-CLASS	279
F.4.1 Record Fields	279

	Page
F.4.2 Operations	280
F.5 CONSISTS-OF-RELATION-CLASS	280
F.5.1 Record Fields	280
F.5.2 Operations	280
F.6 HISTORICAL-ACTIVITY-CLASS	281
F.6.1 Record Fields	281
F.6.2 Operations	281
F.7 Detailed Operation Descriptions	281
F.7.1 Activity	281
F.7.2 Data-Element	284
F.7.3 ICOM-Relation	286
F.7.4 Calls-Relation	288
F.7.5 Consists-Of	289
F.7.6 Historical-Activity	290
Bibliography	292
Vita	296

List of Figures

Figure	Page
1. Iterative Waterfall Life Cycle Model	2
2. IDEF ₀ Hierarchical Decomposition	8
3. Data Flow Diagraming Method	9
4. IDEF ₀ Diagraming Method	10
5. IDEF ₀ Activity Box	11
6. Full-Adder: Entity Declaration	12
7. Full-Adder: Architecture Body	13
8. Target Environment	15
9. SADT Diagram Showing Dominance	35
10. Example SADT Context (A-0) Diagram	36
11. Example SADT Child Diagram	37
12. SADT Tree Structure	38
13. SADT Feedback Loop Types	39
14. An Activity with an Activation Rule	44
15. Template of the Modified Decision Table Orientation	46
16. Heating System A-0 Diagram	54
17. Heating System A0 Diagram	55
18. Heating System A2 Diagram	57
19. Heating System A3 Diagram	59
20. Data Dictionary Entry Format for a Data Element	63
21. Data Dictionary Entry for Combustion_Sensor Arrow	64
22. Compare_Temperature Entity Declaration	65
23. Process Block Declaration for Shut_Off_Furnace	67
24. Complete Architecture Body Definition for Shut_Off_Furnace	68
25. Component Representation for Control_Motor	69

Figure	Page
26. Architecture Body for Control_Motor	70
27. Component Declarations for Control_Motor	71
28. Component Instantiations for Control_Motor	72
29. Complete Architecture Definition for Control_Motor	73
30. Heating System Tree Structure	74
31. Heating System A-1 Diagram	76
32. VHDL Code for the Delay_Five_Min_Activity	77
33. VHDL Test Environment connection to the SADT Model	79
34. Sample VHDL Simulator Output	81
35. SADT Tree Structure for the Lift Problem	84
36. Lift Control System A-0 Diagram	85
37. Lift Control System A0 Diagram	86
38. Lift Control System A1 Diagram	87
39. Lift Control System A14 Diagram	88
40. Lift Control System A2 Diagram	90
41. Lift Control System A23 Diagram	91
42. Lift Control System A-0 Diagram	96
43. Pipeline Arrow Representation	96
44. VHDL Record Type Definition of an SADT Pipeline Arrow	97
45. Pipeline Arrow Using Dot Notation	97
46. VHDL Process Definition for Activity Update_Min/Max	98
47. Revised VHDL Process Definition for Update_Min/Max	101
48. VHDL Code for Adding Timing to the Lift	104
49. VHDL Code for Specifying Initial Conditions in a Record Structure	105
50. Existing Data Dictionary Entry Format for a Data Element	111
51. Proposed Data Dictionary Entry Format for a Data Element	111
52. SADT Transformer Interfaces with the Abstract Model	112

Figure	Page
53. Example SADT Hierarchy	115
54. SADT Model represented as a Generalized List	118
55. SADT Feedback Loop Types	130
56. Lift Control System A2 Diagram using Feedback	131
57. Cyclic Feedback Loop Redrawn	132
58. Example of Refine Source Code	139
59. Example of VHDL Source Code	139
60. VHDL Testbench connection to the SADT Model	142

List of Tables

Table	Page
1. SADT' to VHDL' Mapping	42
2. Decision Table Example – Heater Activity Behavior - A1	45
3. Decision Table Example – Lift Activity Behavior - A21 Part 1	48
4. Decision Table Example – Lift Activity Behavior - A21 Part 2	49
5. Potential Levels of Executability	50
6. Decision Table Example – Heater Activity Behavior - A1	60
7. Applicable Values of Interface Arrows	63
8. Decision Table Example – Heater Activity Behavior - A1	67
9. Incorrect Activity Behavior - A21	82
10. Corrected Activity Behavior - A21	82
11. Activity Behavior - A21 Part 1	99
12. Generic Relationship Representation of Decision Table	120
13. Activity Behavior - A1 (Not Mutually Exclusive)	134
14. Activity Behavior - A1 (Mutually Exclusive)	134
15. Conventionally Oriented Decision Table	134
16. TEST 3 - Combustion Error while Running	140
17. Similarities Between the Refine and VHDL Translations	144
18. Differences Between the Refine and VHDL Translations	145
19. Heater Activity Behavior - A1	152
20. Heater Activity Behavior - A4	152
21. Heater Activity Behavior - A21	152
22. Heater Activity Behavior - A22	153
23. Heater Activity Behavior - A31	153
24. Heater Activity Behavior - A32	153
25. Heater Activity Behavior - A33	154

Table	Page
26. Test Case Key	154
27. TEST 1 - Initial Startup - Phase 1	155
28. TEST 1 - Initial Startup - Phase 2	155
29. TEST 1 - Initial Startup - Phase 3	156
30. TEST 2 - Normal Shut Off - Phase 1	156
31. TEST 2 - Normal Shut Off - Phase 2 (After 5 sec)	157
32. TEST 2 - Normal Shut Off - Phase 3	157
33. TEST 3 - Combustion Error while Running	158
34. TEST 3 - Fuel Flow Error while Running	158
35. TEST 4 - Master Switch Off while Running - Phase 1 (Before 5 sec)	159
36. TEST 4 - Master Switch Off while Running - Phase 2 (After 5 sec)	159
37. TEST 4 - Master Switch Off while Running - Phase 3	160
38. Lift Activity Behavior - A11	161
39. Lift Activity Behavior - A12 Part 1	161
40. Lift Activity Behavior - A12 Part 2	162
41. Lift Activity Behavior - A12 Part 3	162
42. Lift Activity Behavior - A12 Part 4	162
43. Lift Activity Behavior - A12 Part 5	163
44. Lift Activity Behavior - A13	163
45. Lift Activity Behavior - A141 Part 1	164
46. Lift Activity Behavior - A141 Part 2	164
47. Lift Activity Behavior - A141 Part 3	165
48. Lift Activity Behavior - A142	165
49. Lift Activity Behavior - A21 Part 1	166
50. Lift Activity Behavior - A21 Part 2	167
51. Lift Activity Behavior - A21 Part 3	168
52. Lift Activity Behavior - A22 - Part 1	169

Table	Page
53. Lift Activity Behavior - A22 - Part 2	170
54. Lift Activity Behavior - A22 - Part 3	171
55. Lift Activity Behavior - A22 - Part 4	172
56. Lift Activity Behavior - A22 - Part 5	173
57. Lift Activity Behavior - A231 Part 1	174
58. Lift Activity Behavior - A231 Part 2	175
59. Lift Activity Behavior - A231 Part 3	176
60. Lift Activity Behavior - A231 Part 4	177
61. Lift Activity Behavior - A231 Part 5	178
62. Lift Activity Behavior - A231 Part 6	179
63. Lift Activity Behavior - A231 Part 7	180
64. Lift Activity Behavior - A231 Part 8	181
65. Lift Activity Behavior - A231 Part 9	182
66. Lift Activity Behavior - A231 Part 10	183
67. Lift Activity Behavior - A231 Part 11	184
68. Lift Activity Behavior - A232 Part 1	185
69. Lift Activity Behavior - A232 Part 2	186
70. Lift Activity Behavior - A232 Part 3	187
71. Lift Activity Behavior - A232 Part 4	188
72. Lift Activity Behavior - A232 Part 5	189
73. Lift Activity Behavior - A232 Part 6	190
74. Lift Activity Behavior - A232 Part 7	191
75. Lift Activity Behavior - A232 Part 8	192
76. Lift Activity Behavior - A232 Part 9	193
77. Lift Activity Behavior - A232 Part 10	194
78. Lift Activity Behavior - A232 Part 11	195
79. Lift Activity Behavior - A233	196

Abstract

Formalizing an informal requirements specification, such as SADT, and executing the formal specification in a simulation environment, such as VHDL, provides a requirements analyst a means to validate the behavior of a specification early in the development life cycle. This research effort investigated and demonstrated the feasibility and benefit of transforming an SADT specification of a system into an equivalent VHDL executable simulation. Both non-time related behavior and concurrent, real-time related behavior is addressed. First, a decision table extension to SADT is created so that detailed, executable behavior can be specified. Next a mapping from SADT to VHDL is defined. Last, this mapping was applied to two example problems: the Heating System and the Lift (Elevator) Control System. An SADT specification was generated for each of these problems, and the resulting specification was transformed into an equivalent VHDL specification using the mapping technique defined by this research. The VHDL simulation environment was used to execute the specification, determine its behavior, make necessary changes, and re-execute the specification until the proper system behavior was specified. The result was an unambiguous specification which can serve as a formal basis for subsequent design and implementation phases, and ultimately, a product which better satisfies the users needs.

FORMALIZATION AND VALIDATION OF AN SADT SPECIFICATION THROUGH EXECUTABLE SIMULATION IN VHDL

I. Introduction

1.1 Background

The requirements analysis phase of the system development process is the critical first step in the life cycle of a system (Figure 1). A software requirements specification is typically generated in this phase of the product life cycle. The specification is used to describe the "functions the software must perform, the level of performance (speed, accuracy, etc.), and the nature of the required interfaces between the software product and its environment" (10:76). This specification serves as the cornerstone for the remainder of the design and build of the software product; therefore, it is critical that the specification has correctly captured the desired product behavior. According to Levi and Agrawala, the existence of a complete and consistent requirements specification determines the outcome of the remainder of the system's life cycle (30). The ultimate goal of a system developer is to design and implement a system which satisfies the users needs in the most efficient and effective way. Without a solid requirements specification, obtaining this goal is nearly impossible. "No matter how well-designed or well-coded, a poorly specified program will disappoint the user and bring grief to the developer" (37:136).

The process of requirements analysis and the development of a precise requirements specification has been categorized as being "complex and error prone" (28:749). Decomposing a system into smaller functional components is one method used to break the tasks of the requirements analysis phase into manageable pieces. The Air Force Institute of Technology (AFIT) has devoted considerable effort to the development of a computer-aided software engineering (CASE) tool (21) to partially automate the generation of a requirements specification based on functional decomposition. This tool, SAtool II (27, 48), is a graphic-based system using the Integrated Computer Aided Manufacturing (ICAM)

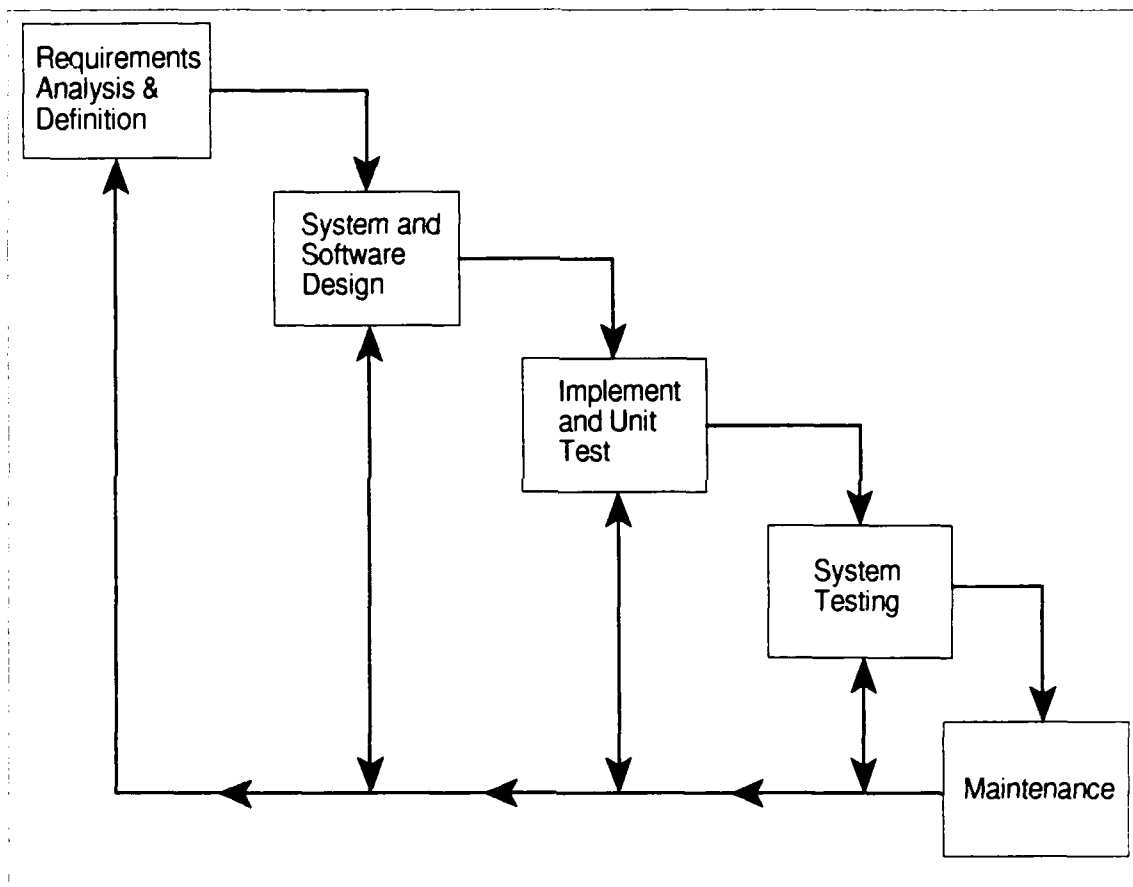


Figure 1. Iterative Waterfall Life Cycle Model (46)

Definition Zero (IDEF₀) subset (33) of the Structured Analysis and Design Technique (SADT¹) specification method. Definitions of SADT and the IDEF₀ method are given in Sections 1.4.1 and 1.4.2. SAtool II produces IDEF₀ diagrams and associated data dictionaries based upon the abstract objects and operations (27) of the system being decomposed. Tevis (48) and Kitchen (27) defined two subsystems in the SAtool II design: the graphical model and the abstract model. The abstract model forms the basis of the requirements specification generated in the analysis and design of a system.

Once the requirements specification has been developed, the contents of the specification must be validated to determine if the specification accurately reflects the system desired by the user (46). Software requirements validation is a software engineering technique used to determine if the written software requirements specification correctly describes the product the client desires.

The validation of a requirements specification is performed by showing the specification meets certain criteria. Boehm (10) defines four basic criteria to be used in this process:

1. Completeness
2. Consistency
3. Feasibility
4. Testability

The *completeness* of a specification is the extent to which the specification is developed. To be complete, the specification must have all the requirements defined. There can be no postponement of requirement decisions in the form of TBDs (to be determined). To be *consistent*, the specification must not contain any requirements which conflict with other defined requirements within itself, or with any other specification defined for the given software system. The requirement is *feasible* if it does not contain any specifications which assume the use or creation of hardware or software which can not be realized using existing hardware and software technology. Finally, the requirements contained in the

¹SADTTM is a trademark of SofTech, Inc.

specification must be *testable*. A requirements statement is testable “if there is reasonable expectation that it can be experimentally shown that a program does or does not satisfy the statement” (4).

The difference between requirements validation and requirements verification is often a point of confusion. The clearest distinguishing feature between the two is the point in the software life cycle when the activity is performed. Requirements validation is performed at the end of the requirements analysis phase of the software life cycle. The primary intent of validation is to ensure the specification explicitly states the users requirements and that a design and implementation based upon this specification will result in building “the right product” (4). In contrast, requirements verification is a continuous process to ensure the steps being taken in the current life cycle phase fulfill the requirements established during the previous phase. Verification is asking at each phase, “Are we building the product right?” (4).

Simulation of the system specification can be an effective method used to validate the specification (46). In the development of digital systems, a comprehensive description language, VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language), has been developed to provide an environment for the specification and simulation of a hardware system design (31). VHDL is used to simulate the specification, thus providing an environment in which the system specification can be validated by the end user. A more detailed definition of VHDL is provided in Section 1.4.3.

1.2 Problem

Given the importance of validating requirements specifications and the availability of the VHDL simulation environment, the problem addressed in this thesis is:

Investigate the feasibility and benefit of mapping an SADT requirements specification into an equivalent VHDL executable specification.

1.3 Research Objectives

In this research, the following objectives were established as the approach used to solve the stated problem. First, determine an algorithm for transforming an SADT specification into a VHDL simulation. Second, manually implement and validate the transformation. Finally, evaluate the contribution to the software design process of simulating system specifications in VHDL.

This investigation will focus on the abstract model representation of the SADT specification generated by SAtool II, and the specification and simulation capabilities available in the ZYCAD VHDL environment. Although the ZYCAD environment is proposed for use in this research effort, the VHDL source code will follow the IEEE Standard 1076-1987 syntax definition (25). Therefore, the source code will be independent of the VHDL environment used.

1.4 Definitions

Several terms have been introduced in the introductory sections of this thesis which warrant further definition before proceeding. These terms are used in the remainder of this document, and their understanding forms part of the background information necessary for the reader.

1.4.1 SADT The Structured Analysis and Design Technique (SADT²) was originated in the late 1970s (13) by Douglas Ross of SofTech, Inc. The technique has evolved over the last several years, and its history can be found in (39, 40, 41). One recent summary of the SADT methodology was written by Marca and McGowan in 1988 (32).

SADT is a methodology to provide a software system designer with a means to develop a specification which completely specifies the product to be developed. Ross (41) uses the analogy of a blueprint or technical drawing used in manufacturing. When a blueprint is created to specify a part to be manufactured, any skilled craftsman can take the blueprint and build the part. The goal of the SADT methodology is to equip the

²SADTTM is a trademark of SofTech, Inc.

software system designer with a language capable of communicating a software system design as completely as a manufacturing blueprint. (39)

The SADT modeling can be done using either a functional approach or a data object approach. The functional approach, called *activity modeling*, focuses on functions and activities during the decomposition of a system. *Data modeling* concentrates on the system data elements and data objects. The IDEF₀ subset of SADT, which is defined in the next section, is based solely on the activity modeling approach.

The remainder of the discussions in this thesis will use the terms SADT and IDEF₀ interchangeably to mean the activity modeling approach. The IDEF₀ method and AFIT specific extensions to the method are best described in Hartrum's technical report (21) and in theses by Kitchen (27), Shyong (43), and Tevis (48).

1.4.2 IDEF₀ IDEF₀ is a definition of a requirements analysis and modeling technique developed for the U.S. Air Force Materials Laboratory in June 1981 by SofTech, Inc. under the Program for Integrated Computer Aided Manufacturing (ICAM). (27, 33)

IDEF₀ is a diagrammatic method based upon a subset of SofTech's Structured Analysis and Design Technique (SADT). ICAM Definition Zero (IDEF₀) is actually a definition of a graphical model and supporting textual descriptions used to represent the requirements of a system. The IDEF₀ model uses a series of hierarchically related diagrams (21) to depict the functional decomposition of the system. This graphical model, with the supporting text, forms the basis of a requirements specification to be used in the later life cycle phases of the system. The primary goal in using the IDEF₀ method is to generate a representation of the system behavior in a form which is easy to comprehend. Then, this graphical representation of the specification is used to explain the complex system specification of behavior to the customer (end user) in a form which is easily understood. Given that the requirements analyst can better convey the specified behavior to the customer, there is a greater opportunity to ensure the final product will meet the actual user requirements.

The generation of an IDEF₀ system model begins by creating a diagram which represents the highest level of abstraction (lowest level of detail). At this level, the entire system

is represented by a single function. This diagram is known as the *environment model* because it represents how the system will interface with the surrounding environment. Next, the environment model is broken down into lower level diagrams which provide a detailed description of the system. This functional decomposition continues down in a hierarchical fashion providing greater and greater detail at each lower level until the system is completely described. Figure 2 depicts this decomposition structure.

At each level, the system is functionally represented by IDEF₀ diagrams. An IDEF₀ diagram is similar to a Data Flow Diagram (DFD), and a comparison of the two methods is depicted in Figures 3 and 4. A DFD diagram consists of *processes* and *dataflows*. Similarly, an IDEF₀ diagram is comprised of *activities* and *interfaces*. Activity boxes represent specific functions the system provides, and the interface arrows represent connections or constraints imposed between the functions. The interconnections between activities can be one of following four types (27), and are represented in Figure 5.

1. *Input Arrow* - enters the left side of the activity box and represents input data which may be needed for that activity to be performed.
2. *Control Arrow* - enters the top of the activity box and represents control information which constrains what the activity performs.
3. *Output Arrow* - outgoing arc from the right side of the activity box which represents data created when the activity is performed.
4. *Mechanism Arrow* - the incoming arc at the bottom of the activity box represents the person or device which will carry out the function being represented.

Activity boxes and interface arrows are then given unique names, which along with a textual description, are used to generate a data dictionary specification for each IDEF₀ diagram. The data dictionary describes the hierarchy of the system by defining parent-child relationships and interconnections for each of the levels of the diagram. This can then be used to recreate the entire system IDEF₀ diagram at a future time.

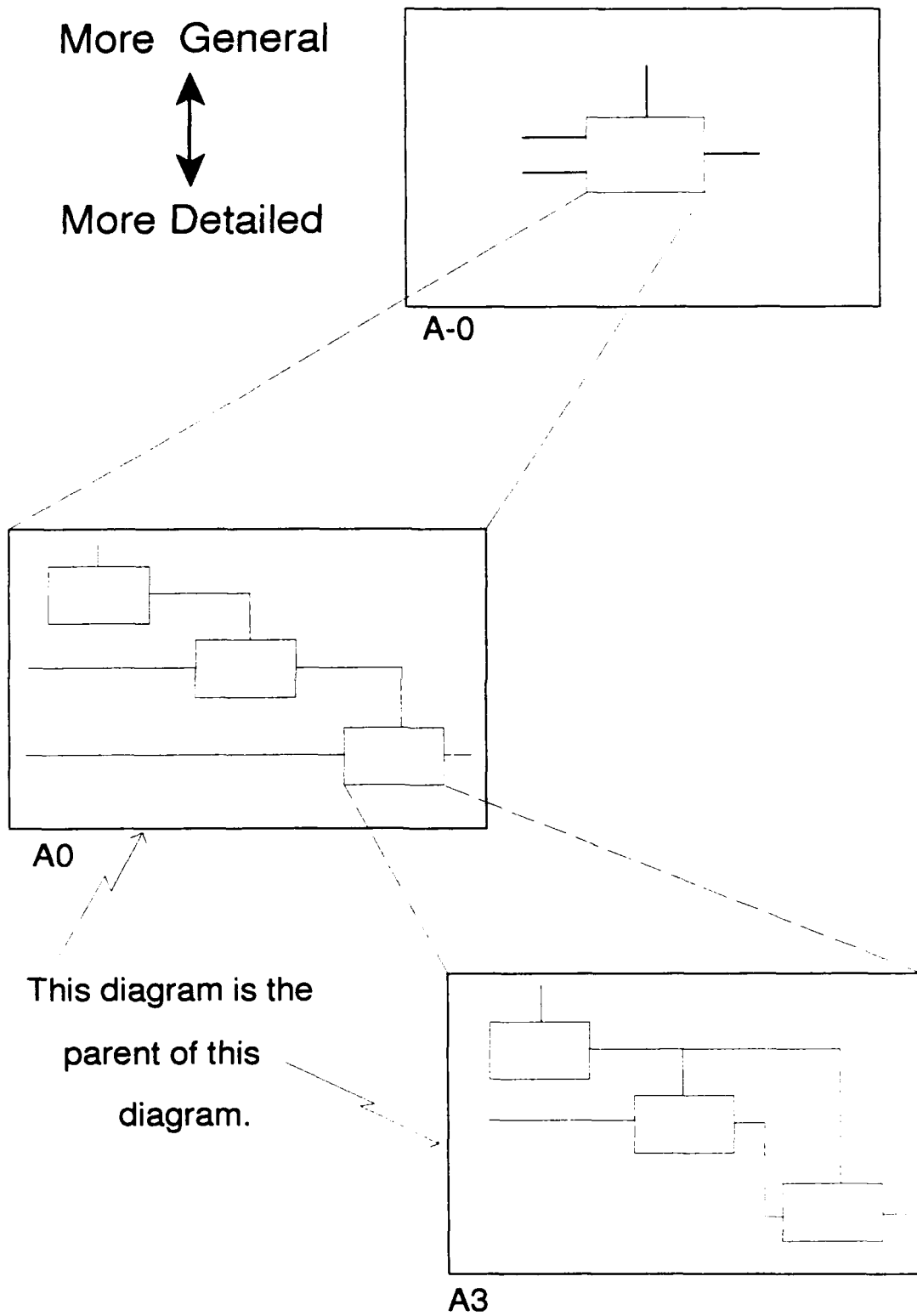


Figure 2. IDEF₀ Hierarchical Decomposition (33)

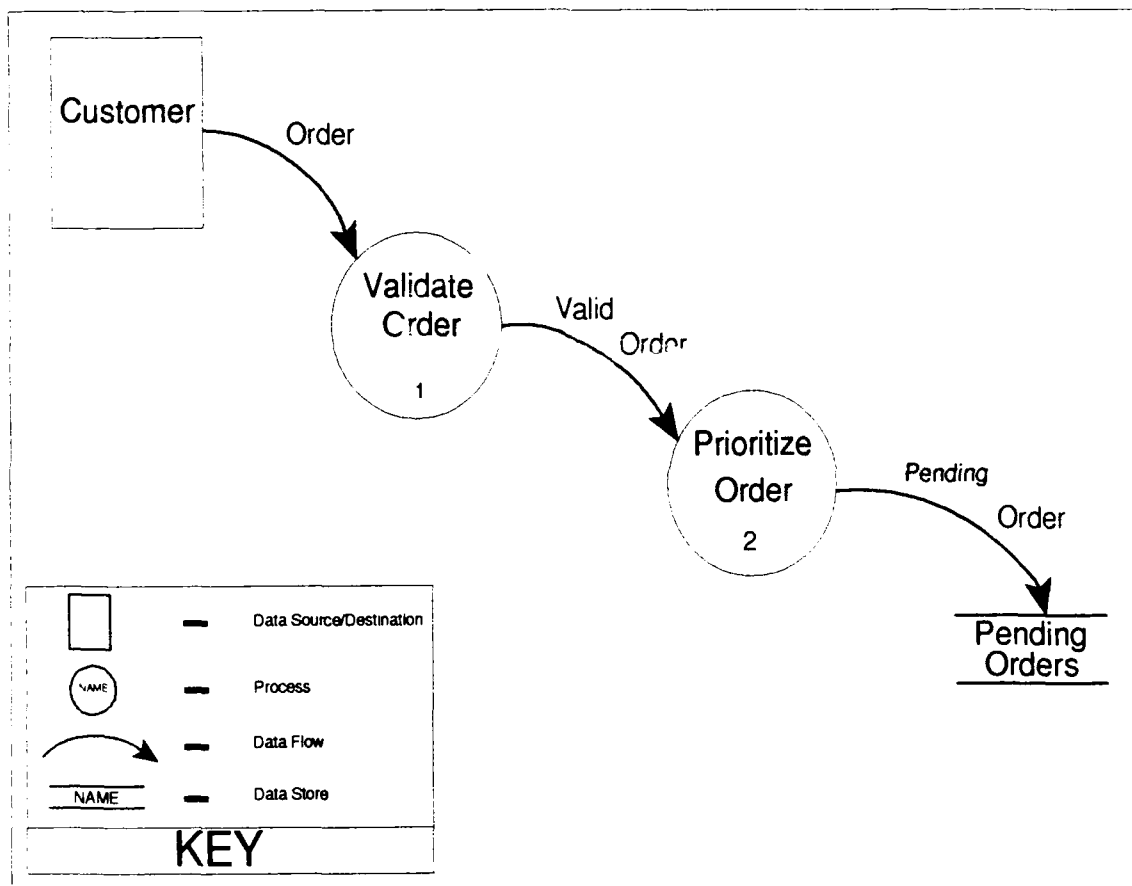


Figure 3. Data Flow Diagramming Method

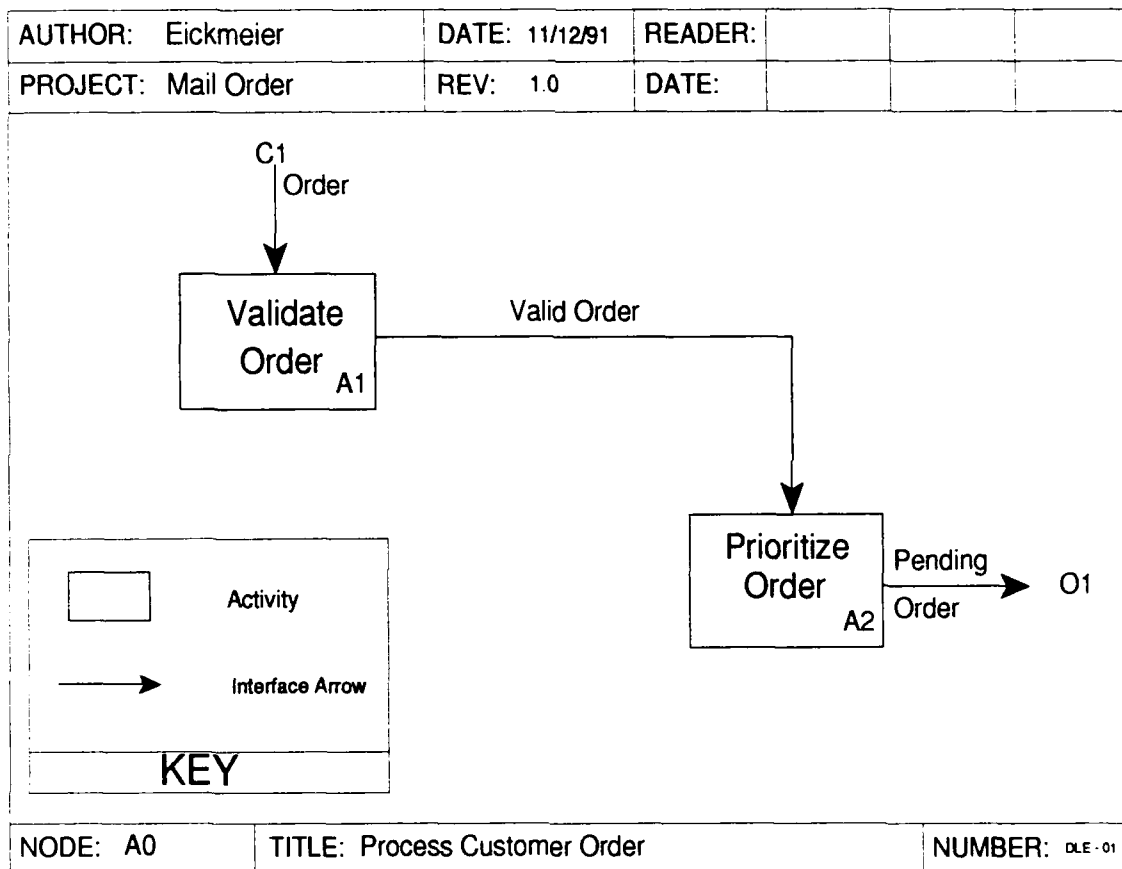


Figure 4. IDEF₀ Diagramming Method

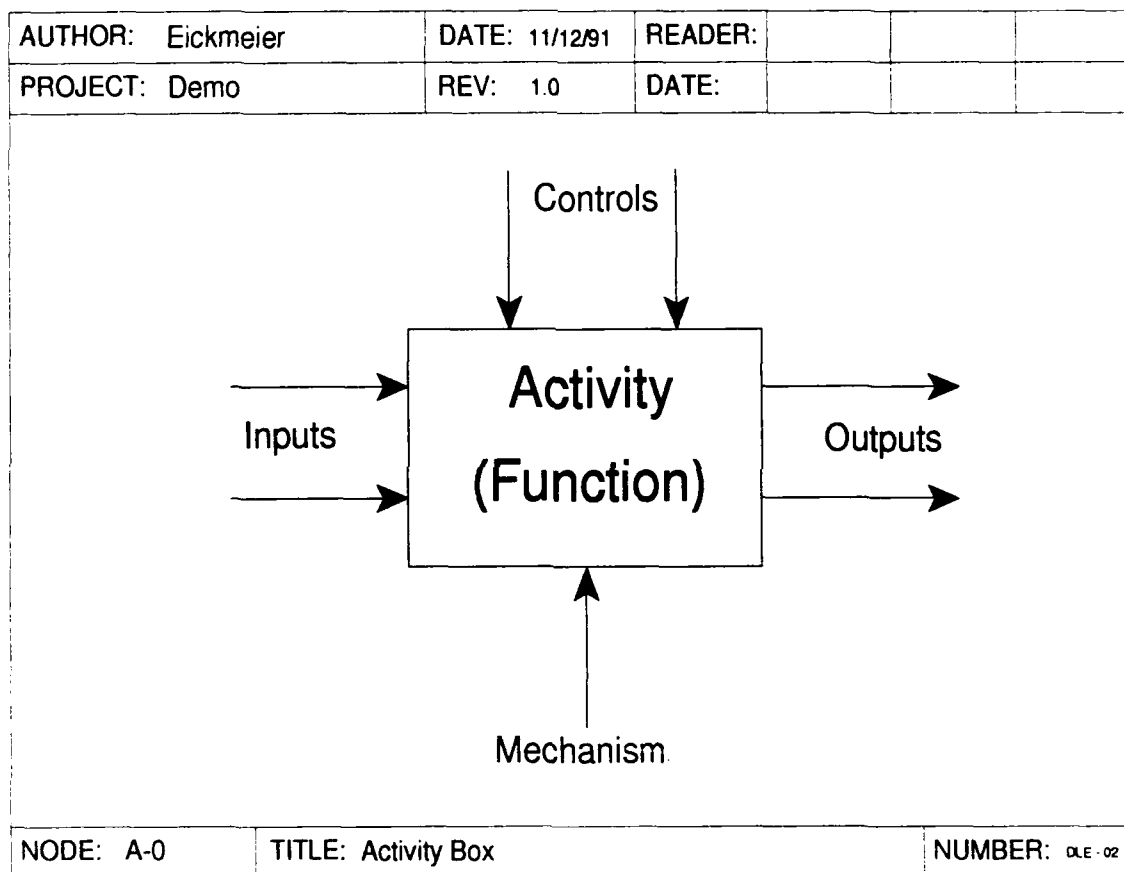


Figure 5. IDEF₀ Activity Box(21)

The use of standard models such as IDEF₀ can greatly aid the design engineer in the development of understandable and complete requirements specifications resulting ultimately in better products.

1.4.3 VHDL The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) was developed to provide a standard representation of hardware designs for the Department of Defense (DoD). The DoD's VHSIC Program Office desired an industry standard description language which was technology independent, yet rich enough to facilitate generation of executable specifications. The goal was to provide the capability to capture a system description at a number of different levels (levels of abstraction), and be able to simulate the system at any mixture of those levels. (31)

The basic unit of description in VHDL is the *design entity* which has two parts: an *Entity Declaration* and an *Architecture Body*. A design entity is used to represent individual components or functions which make-up a system. A design entity can be used as many times as desired in a system description. The *Entity Declaration* defines the inputs and outputs of the entity so other components can interface with it. Figure 6 shows an entity declaration for a Full-Adder. Other design entities may interface with the Full-Adder design entity using the *Ports* A, B, Carry_in, Sum, and Carry_out.

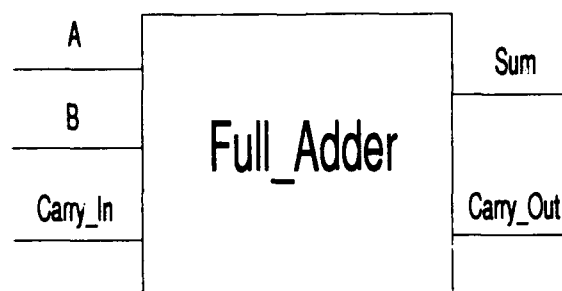


Figure 6. Full-Adder: Entity Declaration (31:22)

The internal representation of a design entity is contained in an *Architecture Body*. This specification can describe the design entity in two ways: 1) in terms of other design

entities, called a *structural description*; or 2) functionally describe the entity's transformation of inputs to outputs, called a *behavioral description* (8, 31). As an example, design entities specified using a behavioral description could be defined for a Half_Adder and an OR_gate. These design entities can now be used to build other design entities using a structural description. A design entity can be described and decomposed using a whole hierarchy of structural definitions, but ultimately, behavioral specification must be created for the lowest level diagrams. The Full-Adder architecture depicted in Figure 7 is an example of using a structural description.

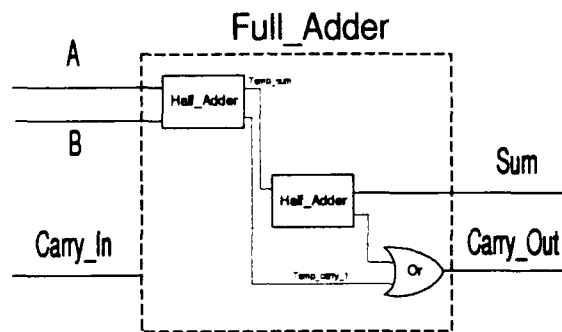


Figure 7. Full-Adder: Architecture Body (31:22)

A detailed description of the VHDL constructs used in this research are defined in Section 3.3. The reader is also referred to two concise summaries of VHDL written by Barton in (7, 8), and the detailed explanation of VHDL by Lipsett, Schaefer, and Ussery in (31).

1.4.4 Refine Refine³ is a formal specification environment offered by Reasoning Systems, Inc. The Refine environment is based upon the Refine wide-spectrum software specification language. The language is termed wide-spectrum because it is applicable over several phases of the software development life cycle. The Refine language is an executable language, and is therefore useful for modeling the behavior of a specification during the requirements analysis phase.

³RefineTM is a trademark of Reasoning Systems, Inc.

The Refine environment was used in a parallel research effort by Douglass (16) in the same manner as VHDL was used in this research. A comparison of the two approaches is contained in Chapter VI of this thesis.

1.5 Scope

This research effort is part of an overall effort to develop an environment for the formal specification and validation of system specifications. The target environment, illustrated in Figure 8, depicts the specifications generated by several requirements analysis tools (SAtool II, Data Flow Diagrams, State Transition Diagrams, and Concept Maps) being transformed into a common representation (9) based on the Abstract Model Manipulator developed in previous AFIT research efforts (2, 3, 27, 44, 48). From this common representation, formal specifications will be generated in both the Refine (16, 29) and VHDL languages. The transformation from the abstract model into Refine was performed by Capt Douglass (16). This research effort specifically addresses the transformation of the an SADT specification into a VHDL behavioral specification. This effort was performed concurrently with the research efforts developing the other parts of the target environment, but was not strictly dependent upon their outcome for successful completion. Significant portions of the SADT analysis for the two representative problems, the Heating System and the Lift Control System, discussed in Chapter IV, the design in Chapter V, and the comparison of implementations in Chapter VI were done as parallel efforts with Douglass (16).

1.6 Standards

Source code developed in this thesis effort will follow the guidelines and standards developed for the AFIT/ENG department (22). In addition, the package and subroutine header formats proposed in Appendix E of Kitchen's thesis (27) will be used as guidelines when appropriate to standardize the header blocks in the SAtool II source code.

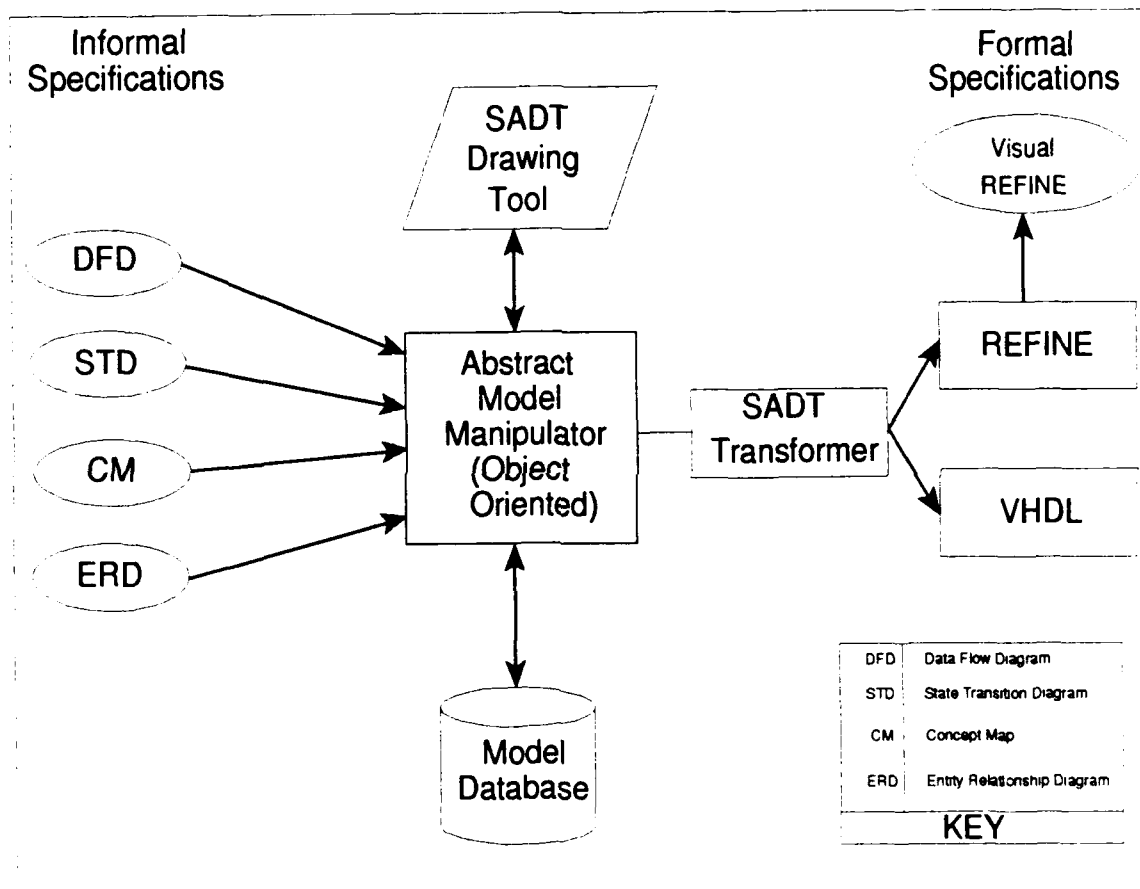


Figure 8. Target Environment

1.7 Equipment and Software

The VHDL behavioral representation was developed using the ZYCAD VHDL environment on the AFIT VAX cluster and the SPARC station network in the VLSI laboratory.

1.8 Sequence of Presentation

The remainder of this thesis is organized into six chapters. Chapter II contains a literature review on current requirements validation methods and tools. Chapter III presents the requirements analysis and design of the SADT to VHDL transformation algorithm. Chapter IV describes generation of an SADT representation of two example problems and the manual transformation of those problems into an equivalent executable VHDL simulation. Chapter V details the design decisions made in specifying an automated transformation method. Chapter VI contains a comparison of the VHDL and Refine implementation methods. Finally, Chapter VII summarizes this thesis effort and provides recommendations of future research to be performed.

II. Literature Review

2.1 Introduction

One of the goals of this research effort was to investigate the benefits of using the VH-SIC Hardware Description Language (VHDL) as a means to validate software requirements specifications. In order to form a basis for comparison of the VHDL validation method, a review of existing requirements validation techniques was performed. This review focused on automated requirements tools which were designed to identify errors in requirements specifications.

For the purposes of this investigation, the automated requirements tools reviewed have been divided into two main categories: Prototyping Tools and Analysis Tools. The reviews of four prototyping tools are contained in Section 2.2. First, the PAISLey executable specification language and its environment is summarized in Section 2.2.1. Next, the STATEMATE specification tool set is outlined in Section 2.2.2, and Section 2.2.3 presents a tool called *The Animator* which uses graphical animation to validate the requirements. The fourth tool, described in Section 2.2.4, is the Refine formal specification environment. Finally, an example of the analysis tool, PSL/PSA, is described in Section 2.3.1.

2.2 Prototyping Tools

In a classic article by Brooks, *No Silver Bullet* (12), the need for automated requirements tools being used to validate user requirements was stated as follows:

I would go a step further and assert that it is really impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product.

Therefore, one of the most promising of the current technological efforts, and one that attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements.

A *prototype software system* is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily

being bound by the same hardware speed, size, or cost constraints. Prototypes typically perform the mainline tasks of the application, but make no attempts to handle the exceptional tasks, respond correctly to invalid inputs, or abort cleanly. The purpose of the prototype is to make real the conceptual structure specified, so that the client can test it for consistency and usability. (12:17)

Summaries of two automated tools which meet Brooks' definition of a prototyping tool follow.

2.2.1 PAISLey Brooks stressed the importance of providing an automated method for the software engineer to demonstrate the behavior of the intended system to the client/user before the actual implementation begins. This capability is provided in the Process-oriented, Applicative, and Interpretable Specification Language (PAISLey) environment. PAISLey was developed in the early 1980s by Pamela Zave initially while at the University of Maryland and subsequently while at AT&T Bell Laboratories (14, 52).

PAISLey is an executable¹ language used in generating requirements specifications for embedded systems (42, 53). Once the specification is complete, the software engineer can use the tools in the PAISLey environment to execute the PAISLey specification. The demonstrated behavior can then be analyzed to determine if the "resulting behavior would mimic the behavior required of the system to be built" (14:247). According to Zave, the "execution of a PAISLey specification is meant to simulate the behavior of the described system rather than to implement it" (53:312).

The PAISLey approach is called an *operational approach* to requirements analysis and specification. This definition is used because PAISLey implements "an executable model of the proposed system interacting with its environment" (52:250). Zave further defines *operational specifications* by stating:

It should be noted that specifications in PAISLey are *operational* (they are implementation-independent models of *how* to solve a problem), rather than *nonconstructive* (statements of *what* properties a solution should have). Operational specifications are inherently lower-level than nonconstructive specifications. (53:324)

¹Executable and interpretable are synonyms, and have been used interchangeably.

To generate a PAISLey specification, the requirements analyst uses a functional programming approach and defines "functions that map inputs into outputs rather than by defining procedures that when executed transform inputs into outputs" (14:248). "The requirements writer decomposes the system under specification and its environment into sets of asynchronous interacting processes" (14:248). These processes can represent data objects, buffers, system functions, and so on. Each process must be defined by specifying the "range of possible states that the process may enter (that is, the state space), and by declaring and defining sets of functions to define how processes change state (successor functions) and interact (exchange functions)" (14:249). Each process maintains a state and "can go through an infinite sequence of discrete changes" (42:199). During the PAISLey execution, the state of the process is evaluated at each process step, and "the current state is replaced by the next state according to the evaluation of the successor mapping using the current state as input" (42).

The PAISLey environment developed by AT&T Bell Labs "provides powerful tools for validating specifications, that is, for making sure that they do what the user needs and for showing they are consistent" (42:204). This environment provides tools for analyzing and executing PAISLey specifications, and consists of a parser, a cross-referencer, a consistency-checker, and an interpreter. The parser reads the textual PAISLey grammar, verifies it for correct syntax, and transforms it into an "internal representation of the specification" (53:314). The cross-referencer performs static analysis of the internal representation and "produces an alphabetic listing of all sets and mappings in the specification, showing where they are declared, defined, and/or used" (53:314). The consistency-checker analyzes the internal representation for internal consistency. If inconsistencies are found, a report is generated detailing the problems. Finally, the interpreter is the interactive user interface into the PAISLey system. It allows the user to enter commands to cause the specification to be executed. During execution, the interpreter displays the execution events to the user's terminal (53).

In summary of the PAISLey specification technique, the following list outlines what various authors see as the strengths and weaknesses of the PAISLey approach.

- Strengths

1. PAISLey provides a “rich set of semantics” which is more expressive than natural language descriptions. This contributes to reducing the “inherent ambiguity” in the requirements specification. (14:296)
2. With the additional expressive power of a requirements specification written in PAISLey, a more formal basis for designing and testing the system is established. (14)
3. Using the set of tools in the PAISLey environment provides the requirements analyst with the means to perform “static structural, behavioral, and protocol error checking” to reduce “ambiguity, incompleteness, and inconsistency” in the specification. (14:297)
4. The PAISLey approach “offers a substantial increase in formality, expressive power, and potential for automation over the current widely known requirements technologies” (52:250).
5. The PAISLey approach makes checking the internal consistency of the specification easy (42).
6. The PAISLey language and environment appears to be unique or at least unusual in having the following desirable features: (53:323)
 - there is both synchronous and asynchronous parallelism free of mutual-exclusion problems
 - all computations are encapsulated
 - specifications in the language can be executed no matter how incomplete they are
 - timing constraints are executable
 - specifications are organized so that bounded resource consumption can be guaranteed
 - almost all forms of inconsistency can be detected by automated checking
 - a notable degree of simplicity is maintained

- Weaknesses

1. “There is no doubt that using PAISLey to represent specifications is harder than using, say, structured systems analysis” (42:205).
2. The PAISLey language is not easy to learn and training is required for it to be used in practice (42).
3. Requirements specifications written in PAISLey are “not intuitive to the non-computer scientist” and therefore are not very helpful for the customers and users gaining an understanding of the specification (14).
4. “A common concern of PAISLey is its design orientation” (14). It encourages the requirements writer to think in terms of the internal behavior (the ‘how’) instead of the external behavior (the ‘what’).

5. Encroaches on Design - Since a PAISLey specification contains such precision and detail, many have reservations about overstepping the requirements phase into the design phase. Zave contends that design does not begin until decisions are made about the allocation and management of physical resources. Using this definition, an executable PAISLey specification can be written without containing any design details. (52:257)
6. Too Much Precision - Because of the precision and formality of a PAISLey specification, it is too technical for the customer/user to understand. Analysts must develop other representations for the PAISLey specification so an untrained person can understand the specification. While this is a disadvantage, it is a small price to pay for the rigor and formality achieved by using this method. (52:258)

In their 1986 article (53), Zave and Schell proposed two potential uses for PAISLey and its environment. The first use, validation, best reflects the original intent of the language. PAISLey can be used to "determine whether the specified system is what the customer wants" (53:323) by executing the specification and demonstrating the behavior of the specification to the customer. The executability of the PAISLey language is the key to this use, and is described by Zave as:

Executability means that a specification can be debugged by the analyst, who may be imperfectly aware of all the consequences of his decisions. For debugging, the interpreter has proven very good, because visibility into the specification is complete (and easily associated with the written specification), and because the analyst has adequate control over the course of execution. (53:323)

Since PAISLey can associate timing analysis with the specification, it can be used to analyze and validate performance requirements as well as functional requirements. "The interpreter can be used as a performance simulator to test whether the timing constraints attached to virtual processes and interactions within the specification lead to externally observable (stimulus-response) timing properties that meet the customer's needs" (53:323-324).

The second use of PAISLey proposed by Zave is in the implementation of the specification in the run-time language. Zave proposes PAISLey be used as part of an "alternative software development life cycle. This new life cycle calls for repeated transformations on the specifications so that they eventually become the code itself" (14:253).

The specific details, including the grammar, of the PAISLey language can be found in Section V and the Appendix of (52) and the PAISLey tools are described in (53).

2.2.2 STATEMATE In an approach similar to PAISLey, i-Logix, Inc. of Burlington, Massachusetts, has developed the STATEMATE² automated requirements generation tool. The STATEMATE environment is founded on the use of *statecharts* which are an extension to finite state machines proposed by Harel to “model complex real-time system behavior unambiguously” (14:230).

When defining the STATEMATE specification environment, Harel (20) identified several necessary properties of a specification language which was going to be used for complex real-time systems. These language properties include: (20:404)

- clear and intuitive
- amenable to generation, inspection, and modification by humans
- precise and rigorous
- amenable to validation, simulation, and analysis by computers
- sufficient depth of semantics to allow use of the language from the initial requirements and specification phase to the prototyping and design phase
- resulting specification forms the basis for modifications and maintenance at later stages

According to Harel, the STATEMATE language and environment “adheres to these principles” (20:404). He further defines STATEMATE as a *visual language* which is based on the following definition:

languages that are highly visual in nature, depending on a small number of carefully chosen diagrammatic paradigms (20:404)

The key difference between the STATEMATE environment and any other graphical based specification tool is the *visual formalism* of the STATEMATE language. Harel defines that

²STATEMATETM is a registered trademark of i-Logix, Inc.

visually formal languages “admit formal semantics that provides each feature, graphical and nongraphical alike, with a precise and unambiguous meaning (20:404).

The visual approach used in STATEMATE is a graphically oriented set of tools enabling a requirements analyst “to prepare, analyze, and debug diagrammatic, yet precise, descriptions of the system under development” (20:403). The analyst prepares the requirements specification in STATEMATE by describing the system using three separate views or perspectives: (14)

- The *Functional View*
- The *Behavioral View*
- The *Structural View*

To represent these three views,

STATEMATE provides graphical, diagrammatic languages, *module charts*, *activity charts*, and *statecharts*, respectively. All three languages are based on a common set of simple graphical conventions and come complete with graphics editors that check for validity as the specifications are developed, and, more importantly, with formal semantics that are embedded within. (20:405)

The *module chart* depicts the structural view of the system in a manner similar to data flow diagrams. The analyst performs a hierarchical decomposition of the development system “into its physical components, called *modules* here, and identifies the *information* that flows between them” (20:404). The resulting diagram is the first of the three graphical languages of the STATEMATE system. The remaining languages, *activity-charts* and *statecharts*, are described by Smith and Gerhart as follows: (45:50)

Activity charts depict the functional characteristics of the design. An activity chart provides a static view of the system functions, data interfaces and control interfaces. The graphical entities of the activity chart are internal activities (sharp cornered boxes), which can be operations, objects, or functions, depending on the particular method chosen for design; control activities (shaded round edged boxes), which represent the behavioral aspects of the design; data flow

(solid lines), which represent data items in the system; and control flow (dashed lines), which represent conditions, information flow, and events in the system.

Statecharts are the control mechanisms for the activity charts. They illustrate the dynamic behavior associated with the functional parts of the system. Specifically, they describe how and when activities become initiated, interrupted, or terminated. The graphical entities of a statechart are states (round edged boxes) and transitions (solid arcs). The statechart is an elaboration of the control activity on an activity chart. It is the mechanism by which control activities are specified.

STATEMATE maintains a central database common to all three graphical editors. Because of the desire to “enable the user to run, debug, and analyze the specifications and designs that result from the graphical languages” (20:405). The database was “constructed to make it possible to rigorously execute the specification”(20:405). The common database also provides an analyst the capability to retrieve a variety of information from the STATEMATE environment (20).

In addition to the information from the three graphical languages, the database maintains a data-item description for each element in the STATEMATE description. This description is stored in a *forms language* in which the analyst can enter information which is nongraphical in nature. This information is represented as a form, and contains entries such as: (20:409)

- Name
- Synonym
- Description
- Definition - define compound events and conditions
- Type and Structure
- Consists of - relations which allow the structure of data items into components
- Attribute Name and Attribute Value - to associate attributes with the data item

This information can be “viewed and updated not only from the special forms editor, but from the appropriate graphical editor as well” (20:409). The STATEMATE system can

perform consistency and completeness checks on the database information to ensure the information entered in one editor corresponds to information entered in another editor.

Once the specification has been formed in the appropriate graphical editors, the analyst can benefit from the formalness of the STATEMATE language via simulation of the specification. The simulation of the STATEMATE specification gives the analyst insight into the dynamic behavior of the system. The simulation is displayed graphically to the analyst as it executes, and changing colors on the STATEMATE charts show the execution progress. According to Harel, "The result is a visually pleasing discrete animation of the behavior" (20:411).

Smith and Gerhart identify the significance of the simulation feature of STATEMATE:

The primary importance of the simulation feature appears to be the animation of specifications for the purposes of demonstrating to the customer the significant features of a complex system design. (45:54)

Smith and Gerhart used STATEMATE to model the cruise control problem. They describe their experience with the simulation feature of STATEMATE:

The simulation features of STATEMATE uses a series of simulation steps where each step modifies a collection of one or more states, events, conditions, and data items. The designer conducts a simulation by modifying or generating events and conditions, which are depicted as the static control flow inputs to a statechart. The transitions in a statechart are enabled by generating events and causing conditions to become true. Some transitions may themselves generate new events, change data items, or change conditions. The transitions cause new states to be entered that may correspond to particular activities (functions, objects, or operations) being activated or deactivated. STATEMATE graphically displays the current state of the simulated statechart. The currently active states are highlighted, as are the transitions between states. With the behavioral approach, all the control elements of a design are placed in a single statechart so that the effect of simulation on the entire system can be observed. (45:51)

As Smith and Gerhart described, the execution of the STATEMATE simulation is based on transitions between states of the system. The simulation environment "runs" in

a step-by-step manner, where a step is defined as "one unit of dynamic behavior" (20:410). At the beginning and the end of each step, the system being simulated is in some legal *status* which contains the current values of the active states, variables and conditions (20:410).

The ability to model real time constraints in the STATEMATE simulation environment is incorporated into the statechart diagram of the system (45). The statecharts specify the control information of the system, and can contain time-dependent control specifications. Smith and Gerhart (45) provide an example use of this feature in their cruise control problem. Their cruise control system needed to meet a constraint that "the system must disengage cruise control within 1 second of engaging the brake" (45:53). They express this constraint in a statechart called *Speed-control* using a transition **timeout (brake-on, 1 sec)** from a system state *Cruise-active* into a system state *Error* (45). This syntax specifies that the state of *Speed-control* will transition from the *Cruise-active* state to an *Error* state if 1 second has expired since the **brake-on** event has occurred.

In a case study performed by the Software Engineering Institute (SEI) in 1988, SEI found STATEMATE "well suited for modeling software processes" (26:78). They used the commercially available version of STATEMATE to model the "process used to modify large volumes of technical documentation to correspond to software changes for the F-16 aircraft's software systems" (26:78). They conclude that, "STATEMATE offers a representation formalism that is highly visual, yet formally defined" (26:78) where they were

able to easily simulate the process' behavior in an interactive, animated fashion. This capability has proven to be highly valuable in validating the model and examining its robustness to changing conditions. (26:78)

The STATEMATE environment is not limited to use in software designs. In the most recent release of STATEMATE, the STATEMATE system will generate IEEE-1076 standard VHDL source code from the graphical STATEMATE specification (49). This addition was made to "enable hardware designers to use STATEMATE not only for the specification and early design stages, but also for the later [integration and testing] stages (20:412).

In summary, the STATEMATE approach appears to possess the same strengths as the PAISley approach defined in Section 2.2.1. It also overcomes many of the PAISley weaknesses related to the difficulty of working with the textual-based environment because STATEMATE is visually-oriented which is easier to learn, and understand. In addition, the ability to specify real-time constraints and the generation of VHDL source code allows STATEMATE to represent a broader class of problems than PAISley.

2.2.3 The Animator The Department of Computing at London's Imperial College has investigated the use of animation as a method for validating requirements specifications (28). Animation of requirements specifications meets Brooks' definition of a prototyping system since it helps the "clients and analysts visualize the behaviour (sic) of the specified system" by allowing them to walk through the specification visually (28:750). This approach goes beyond the static validation which can be done by reading a written document. It allows the client and analyst to gain an understanding of the specification's dynamic (or execution) behavior through the animation. (28)

Animation techniques are less exact and detailed than executable specifications, such as PAISley discussed in Section 2.2.1. Kramer and Ng claim this is deliberate and is a benefit of this technique. They explain,

Executable specifications require precise action definitions and more mathematical sophistication than is usually available from analysts or their clients. On the other hand, prototyping requires the inclusion of more implementation detail. Hence, although both can potentially provide a more exact execution of the specification, they require far more information. We feel that this is inappropriate to requirements specification, where such formality and detail may actually obscure the more general requirements that are being specified. That is not to say that we do not believe in those approaches, but rather that they could be used in later phases of system specification. Animation seems to provide the right balance for this level of requirements specification and for obtaining a reflection of its intended behaviour (sic). (28:750)

Based upon their beliefs in the merits of using animation, members of Imperial's Department of Computing have created an entire requirements analysis environment on

an Apple Macintosh. They have named this system *The Analyst Workstation*. The workstation provides an “interactive software tool” which supports two subsystems: (28:766)

- *The Analyst* - This subsystem implements a requirements analysis and specification technique called CORE which is a graphical based method supporting functional decomposition of the system. *The Analyst* subsystem also includes “rule-based syntactic and semantic checking” of the CORE diagrams, and the corresponding CORE specification database produced during the specification phase. (28:752)
- *The Animator* - This subsystem provides a graphical user interface to perform and analyze animation of the system specifications. Animation is based upon the CORE specification generated in *The Analyst*, and is

... based upon the notions of transactions: collections of actions which represent some scenario in the specification. Transactions are generally selected by a process of action-by-action browsing and animation. Action definitions are given by sets of mappings from inputs to outputs, aided by built-in functions and user-defined functions. Control of action execution is given by simple rules based on data arrival. Facilities are also provided for transaction replay, with data modification if desired. (28:766)

According to Kramer and Ng, *The Analyst Workstation* supports the three primary activities of the requirements analysis phase which they define as: (28:749-750)

1. *Requirements Elicitation* - The process of collecting together, from the client, all the facts relating to the required system.
2. *Requirements Specification* - The process of producing a clear, consistent and reasonably complete definition of the client's requirements.
3. *Requirements Validation* - A process to detect errors or “failures of understanding” contained in the Requirements Specifications. The simplest form of validation is identification of inconsistencies within the specification. The second and “more complex validation is that which tests whether or not the specified behavior is as required by the end user/client.” (28:767)

Applying animation techniques to the requirements validation phase can help overcome the "barriers of understanding" between the requirements analyst and the client. It is these barriers which lead to errors ("failures of understanding") in the requirements specification. If these errors can be detected and corrected early in the development cycle, the high cost of the "widespread repercussions" in the later design and implementation phases can be reduced.

2.2.4 Refine Refine was introduced in Section 1.4.4 as a formal specification environment which can be used to model system behavior. In an effort which paralleled this research, Refine was used by Douglass (16) to investigate the benefits of executing requirements specifications. This section provides a more detailed discussion of the Refine environment used by Douglass.

Refine is characterized as a *wide-spectrum* language because it is "capable of expressing very-high-level specifications of programs, low-level (or target-level) programs, and partially refined specifications that contain a mixed range of constructs from very-high-level to low-level (38:1-4)." Two main components of the Refine environment provide the expressive power: the Refine specification language and the Refine object base (38).

The Refine specification language is used to manipulate the object base, and by so doing, models a systems behavior. The Refine specification language supports three programming styles: procedural, transformational and logic-based (38).

The procedural constructs include assignment, if-then-else, while loops, enumerations over sets and sequences, and sequential compositions of statements (blocks). The transformation constructs include a "transformation rule" construct and a powerful pattern language for specifying "program schemas". The transformation constructs are particularly useful for writing program transformations; however, they are sufficiently general that they are useful in other Refine programs as well. The logical constructs are the usual first-order logic operators: and, or, not, implies, and universal and existential quantification. (38:3-2)

The Refine specification language is used to specify a system's behavior, and the Refine environment provides facilities to execute this specification. During execution, the

specification can easily be modified and re-executed until the desired system behavior is achieved. This capability provides the benefits of a rapid-prototyping system in validating the requirements specification.

The Refine Object Base maintains information about the system being developed, including detailed information about the specifications themselves. Several facilities are provided in the Refine environment for recognizing and manipulating objects in the object base. These facilities allow the Refine environment to transform the formal specification from high level requirements into detailed software specifications, and even to implementation if desired. This ability of the Refine environment to generate software specifications has potential use in future research of automating the transformation of an SADT specification into a VHDL specification and is discussed further in Section 7.4.

2.3 Analysis Tools

The prototyping tools discussed in Section 2.2 provided methods to execute the requirements specification to support requirements validation. This section describes an analysis tool, PSL/PSA, which provides an automated tool for syntactically analyzing the requirements specification. The syntactic analysis is the method used to validate the requirements specification.

2.3.1 PSL/PSA One of the earliest requirements analysis methods, PSL/PSA³, was developed in the late 1970s at the University of Michigan at Ann Arbor by Professor Daniel Teichrow (14, 15, 18, 47). The PSL/PSA method is founded on "three primitives: objects, attributes, and relationships" which Davis claims is "similar to the well known entity-relationship model" (15:473). The automated tool developed for the PSL/PSA method is completely textual based and is comprised of two distinct entities:

- Problem Statement Language (PSL)
- Problem Statement Analyzer (PSA)

³PSL/PSATM is a registered trademark of Meta Systems, Inc., Ann Arbor, Mich.

The PSL portion of the tool is a "high-level, non-procedural language" which is used by the requirements analyst to formally describe the system "requirements and functional specifications" (15:473). Defining the system using this well-defined description language provides a common representation of the design information. This facilitates better communication between the analyst and the designer resulting in a better understanding of the problem by both the requirements analyst and the customer. (18)

The biggest benefit of having a rigorously defined specification language is the ability to use it in a computer-aided development environment. The computer can be used to manage and analyze the large amounts of requirements information generated during the analysis and documentation of the system functional specifications. (18, 47) The tool developer states, "The objective of PSL is to be able to express in syntactically analyzable form as much of the information which commonly appears in Systems Definition Reports as possible" (47:44).

The analysis and extensive report generation capabilities of the Problem Statement Analyzer (PSA) portion of the PSL/PSA is what has kept this method alive since its inception in 1977. (14)

PSL performs three primary tasks for the system analyst:

1. *PSL Description Analysis* - PSA processes the PSL description statements and can "detect errors such as omissions and inconsistencies" (15:473). The analyst can request PSA to examine the PSL description for similarity and consistency of the inputs and the outputs of the system being defined, or to detect unused data objects, or to locate gaps in the flow of information between objects. (47)
2. *Generation and Maintenance of a Development Information Database* - PSA provides the capabilities to record, modify, and analyze PSL system descriptions in a computerized database. (18, 47)
3. *Automated Report Generation* - PSA provides a report generation capability to generate a "System Definition Report" which "includes a large amount of material produced automatically from the data base" (47:44).

The use of the PSL/PSA Report Generator can save the requirements analyst significant amounts of time in writing requirements specifications. The computer-generated requirements documentation is normally higher 'quality' than what is produced in a manual system. Teichroew claims the added 'quality' is one of the major benefits of his method. (47) He substantiates this by insisting,

The "quality" of the documentation, measured in terms of preciseness, consistency, and completeness is increased because the analysts must be more precise, the software performs checking, and the output reports can be reviewed for remaining ambiguities, inconsistencies, and omissions. (47:46)

The PSL/PSA method is best suited to the management information system domain and still holds some popularity in this area. Davis reports that the method has been used on "numerous large system [designs] for the DoD," and serves as the design tool for AT&T Long Lines' "information system development efforts." (14:95) He also recounts of some recent enhancements to the tool which have been made by Meta Systems of Ann Arbor, Michigan. These improvements added an interactive graphical front end, called the Structured Architect⁴, to replace the textual method for entering the analysis data. (14)

The PSL/PSA method, especially the Analyzer feature, offers some potential for realizing the benefits of validating requirements early in the software development life-cycle. The identification of ambiguities, inconsistencies, and omissions early, before further development begins, will reduce the high costs of fixing these errors if they are not found until after the software has been delivered.

2.4 Summary

This literature review has briefly covered four Prototyping and Analysis Tools which have been used as a means to perform validation of software requirements specifications. Refine offers many of the same benefits as VHDL in the validation of SADT requirements specifications. It has been used in the remainder of this thesis as the basis for comparison. Chapter VI summarizes the similarities and differences of using Refine versus VHDL.

⁴Structured ArchitectTM is a trademark of Meta Systems, Inc., Ann Arbor, Mich.

III. Requirements Analysis of an SADT to VHDL Transformation Algorithm

3.1 Introduction

The objectives of this research, as stated in Chapter I, were to:

Determine an algorithm for transforming an SADT specification into a VHDL simulation; manually implement and validate the transformation; and evaluate the contribution to the software design process of simulating system specifications in VHDL.

This chapter addresses the first goal of the problem statement. The following sections 1) detail the primary attributes of the SADT method; 2) specify the VHDL constructs which support the SADT attributes; 3) summarize how SADT maps to VHDL; and 4) discusses the level of executability of the resulting VHDL simulation.

3.2 Definition of the SADT Method's Primary Features

Definitions of SADT and IDEF₀ were presented in Section 1.4. These definitions are expanded further in this section to highlight the semantics and features of the SADT language which were addressed in the SADT to VHDL translation process.

3.2.1 SADT Building Block Components An SADT diagram is built from two principle components: boxes and arrows. The boxes represent the *Activities* or functions of the system, and the arrows represent *Interfaces* between the boxes. Section 1.4.2 defined the four arrow types: input, control, output, and mechanism; and their respective placement on a box: left, top, right, and bottom. The following syntax rules for activities and arrows were summarized by Hartrum (21) and are key elements in determining the mapping from SADT to VHDL.

1. Each activity must have at least one control arrow and one output arrow.
2. An Activity box may represent a collection of related functions, not all of which are operating at a given time. Each function of an activity box represents a process that may execute simultaneously with other processes.

3. Arrows represent data or information needed or produced by an activity. They should be viewed as constraints on the activity, not be viewed as flows or sequences of operations.
4. Any given activity can proceed when and only when the data or information inputs and/or controls are present or available. The activity may not need to use the value of the data or information. The mere occurrence of an input or control may be all that is required. Also, the presence of "all" inputs and controls may not be needed for the activity to proceed. Marca and McGowan (32) describe a notation called *Activation Rules* which allow the SADT designer to specify what inputs and controls are needed to generate a particular output. Activation Rules will be described further in Section 3.5.1.
5. Arrows can represent a single data item or a collection of data items called a *Pipeline*. Pipelines and other special arrow notation are described in Section 3.2.3.

3.2.2 SADT Diagrams and Models Ross states the basic premise of the SADT method as: "The human mind can accommodate any amount of complexity as long as it is presented in easy-to-grasp chunks that together make the whole (40:25)." This premise drives SADT's hierarchical decomposition of the system into *levels*. Each lower level contains more and more detail. The hierarchical nature of SADT was depicted in Chapter I, Figure 2.

A level in an SADT decomposition is contained in a *Diagram*. An SADT diagram is a collection of boxes and arrows which describe a specific portion of the system. A diagram usually contains three to five activities arranged from left to right in *dominance* order. An activity has a higher dominance if it generates outputs which are used to control other activities in the diagram. The activity boxes are placed on the diagram with the highest dominant boxes on the left followed by lower and lower dominant boxes on the right. The SADT diagrams shown in this document follow dominance ordering. As an example, in Figure 9 the Validate Order activity (A1) generates an output (Valid Order) which controls the Prioritize Order (A2) activity. Thus, Validate Order (A1) has a higher dominance than Prioritize Order (A2).

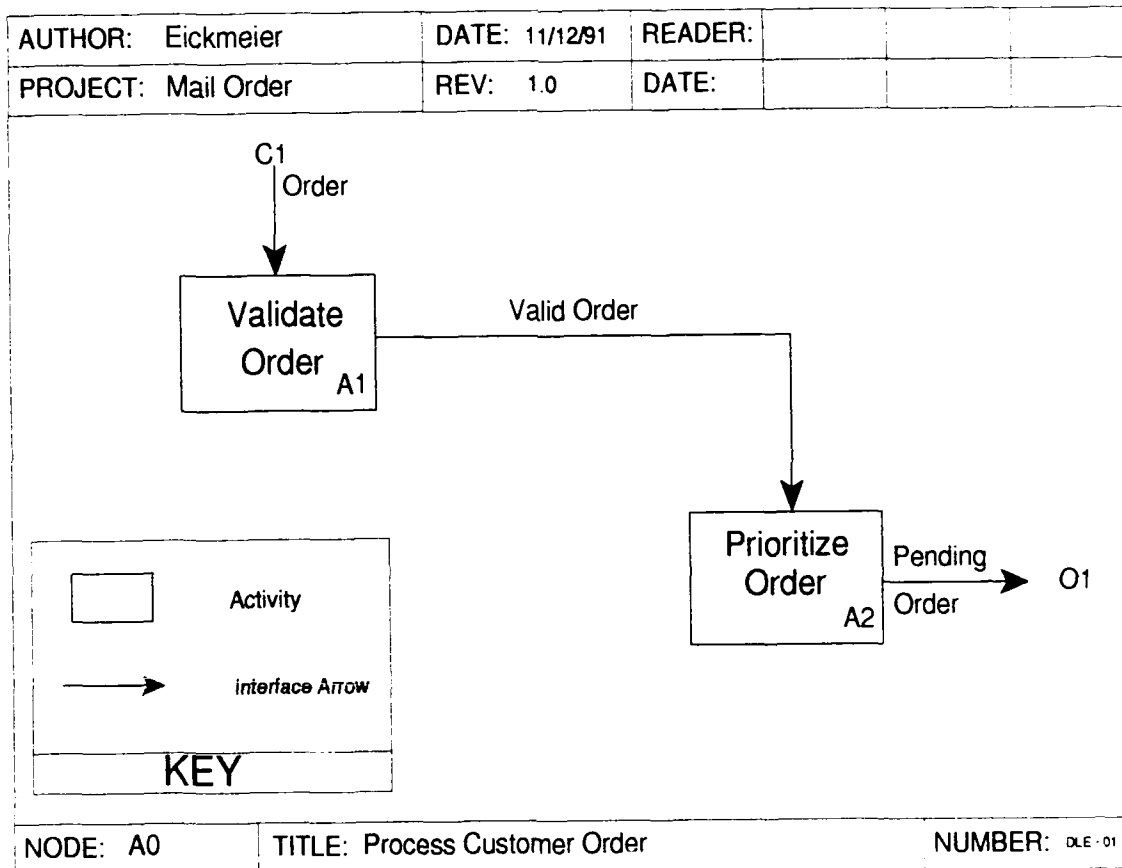


Figure 9. SADT Diagram Showing Dominance

An SADT *Model* is a collection of SADT diagrams which describe a complete system. Marca and McGowan define an SADT Model as: "a collection of carefully coordinated descriptions, starting from a high-level description of the entire system and ending with detailed descriptions of system operation" (32:9).

The model is a hierarchically organized set of diagrams. Each diagram describes a set of activities or functions within a specific boundary of the system. The boundary starts at the highest level, the A-0 diagram, which describes the boundary between the system being described and the external environment. This diagram is called the *Context Diagram* of the model, and it shows the inputs, controls, outputs, and mechanisms coming from and going to the external environment. Figure 10 is an example of an A-0 diagram.

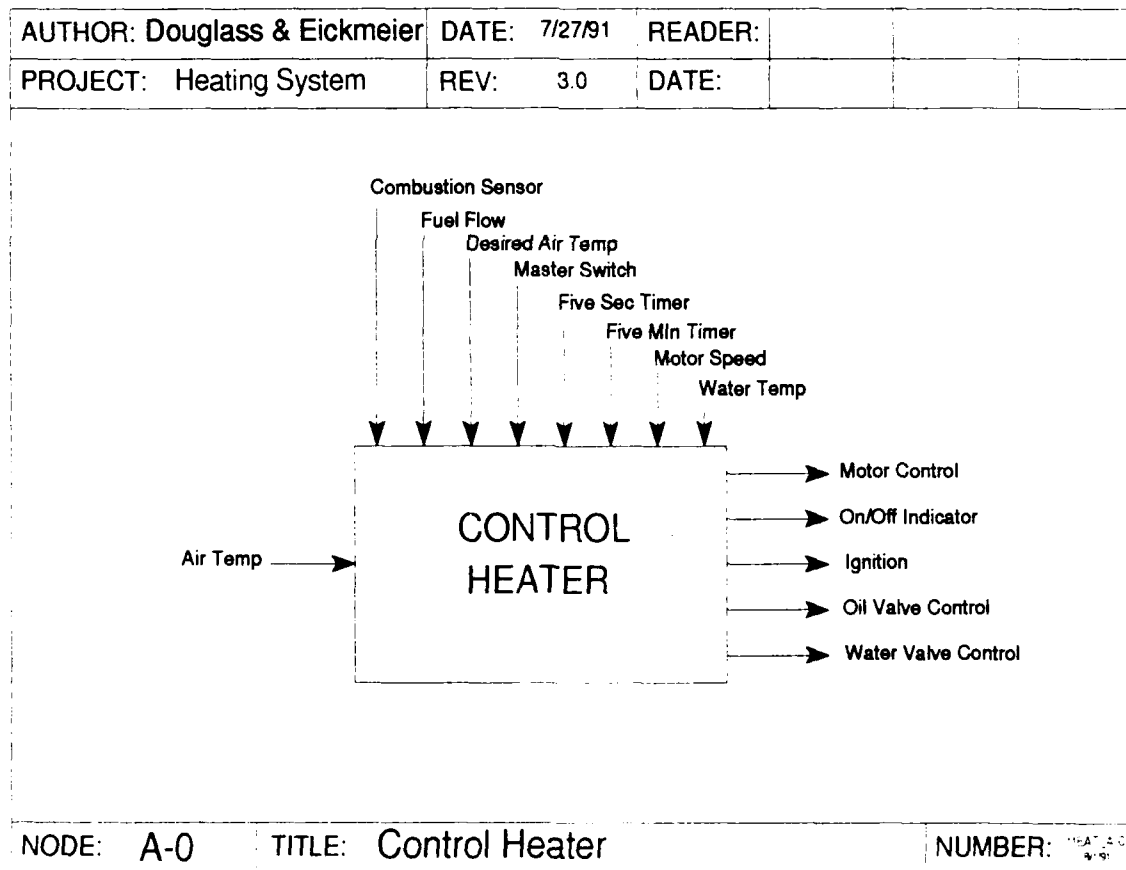


Figure 10. Example SADT Context (A-0) Diagram

At any level, an activity describes a specific portion of the system. The activity is

bounded by the Inputs, Controls, Outputs, and Mechanisms (*ICOMs*) which touch the box. This activity can be further decomposed in a lower level diagram to provide greater detail. An activity which is further decomposed is called a *Parent* activity and the activities at the lower level of decomposition are called *Child* activities. The parent A-0 diagram shown in Figure 10 is decomposed in the A0 diagram contained in Figure 11. This A0 diagram is termed a *Child Diagram* of the A-0 diagram. The ICOMs of the parent activity define

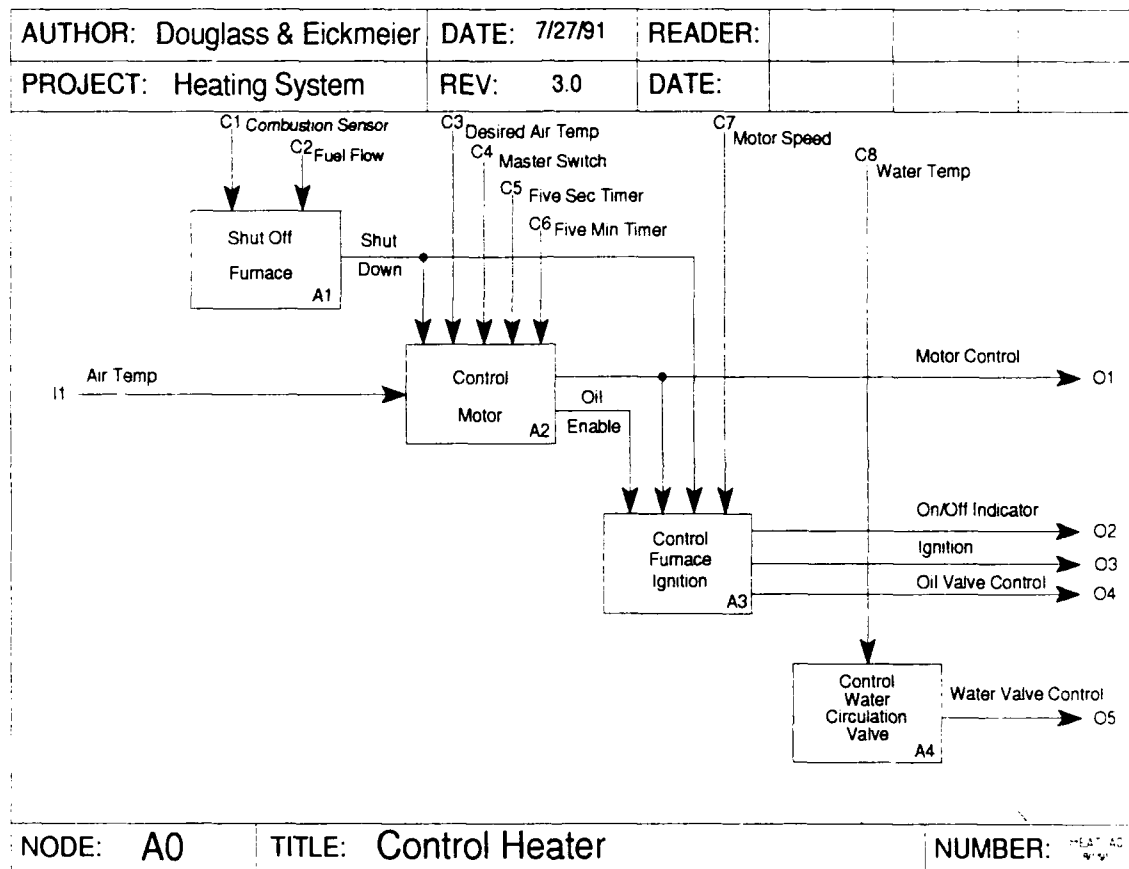


Figure 11. Example SADT Child Diagram

the *Boundary Arrows* of the child diagram. These boundary arrows, the arrows connected only on one end, are numbered in the child diagram according to the corresponding parent box edge. The numbers, called *ICOM Codes* are generated from top to bottom for inputs and outputs; and left to right for controls and mechanisms. The child diagram must have exactly the same boundary arrows as the ICOMs of its parent activity. In the A0 diagram

(Figure 17) shown, the boundary arrows are labeled with ICOM codes I1, C1 through C8, and O1 through O5.

The terminology and hierarchical layout of an SADT model is based upon the notion of a tree-structure. An SADT model can be thought of as a tree-shaped collection of diagrams. The A-0 diagram (root diagram) being the most general and the lowest level diagrams (leaf diagrams) being the most detailed. The activities represent nodes in the tree, and are numbered in node order. A tree structure representation of a typical problem is shown in Figure 12. The SADT decomposition process is considered complete when the lowest level diagrams containing the leaf node activities describe the entire system in an easy to understand representation (19).

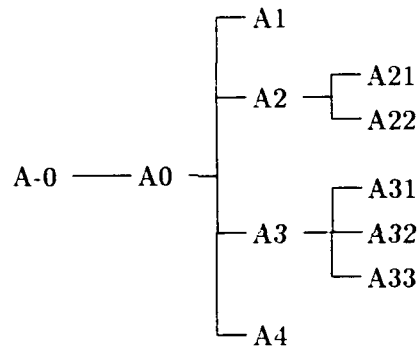


Figure 12. SADT Tree Structure

3.2.3 Special Diagram Notations for Arrows The SADT designer may make use of several techniques to simplify the description of systems, and make diagrams as simple and uncluttered as possible. This research has limited the number of notations included in the SADT subset used in the mapping of SADT to VHDL. The following concepts were included in the SADT subset.

- **Branches** – An arrow may *branch* into two or more arrows, each having the same name and carrying the same information.
- **Pipelining** – A *Pipeline* is a single arrow which represents a collection of sub-arrows entering or leaving an activity.

- **Feedback Loops** – The SADT methodology allows the concept of feedback to be used in the modeling process to represent iteration or recursion. There are two forms of feedback in SADT.

1. *Cyclic Feedback* where the output loops back to the input or control of the same activity.
2. Where an output loops back to an activity of higher dominance. This form of feedback can be either:
 - (a) *Dataflow Feedback* - the output loops back to a previous input.
 - (b) *Control Feedback* - the output loops back to a previous control.

Examples of these feedback types are shown in Figure 13.

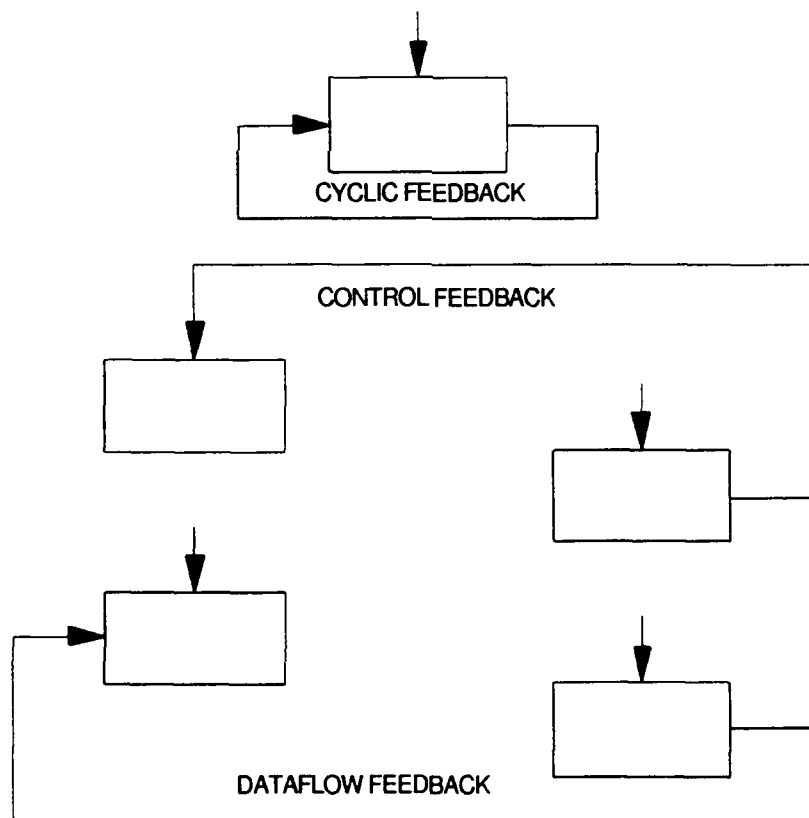


Figure 13. SADT Feedback Loop Types

3.3 Definition of VHDL's Primary Attributes

In this section, definitions of the subset of VHDL constructs used in the mapping from SADT to VHDL are presented. The definitions combine information from the IEEE Standard VHDL Language Reference Manual (25) and the book by Lipsett, Schaefer, and Ussery (31).

3.3.1 VHDL Building Block Components

Design Entity The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. Section 1.4.3 stated that a design entity has two parts: an *entity declaration* and an *architectural body*.

Entity Declaration Defines the entity's interface to the external environment; it specifies the *ports* of the entity in which data may flow in and out. Formally, ports may have a mode **in**, **out**, **inout**, and **buffer** for indicating the flow of data. Modes **in**, **out**, and **inout** are self-explanatory. The mode **buffer** is not used in this research and the reader is referred to (25) and/or (31) for a complete description of this mode.

Architectural Body The description of the internal behavior or structure of a design entity. A structural description is used when the design entity is further decomposed into lower level entities, and it describes how these entities are connected together. A behavioral description is used at the lowest level of decomposition and contains a description of how the entity's inputs are transformed into outputs.

Interface Object An *interface object* provides a channel of communication between design units. *Signals* and *Ports* are examples of interface objects. These two instances are the ones used in the SADT to VHDL mapping.

Signal An object which holds a value. The signal actually holds a series of values, both present and future values.

Port A signal declared in the interface list of an entity declaration. See the above definition for an entity declaration for more details.

3.3.2 VHDL Blocks and Models

Block A *block* is a collection of design entities which represents a portion of the whole design. A hierarchy of blocks is often used in functional decompositions of a system. Blocks in VHDL execute concurrently.

External Block The top-most block in a hierarchy is called the *external block*. This block is the design entity itself, and it defines the interface of the design entity to the external environment.

Process Block A *Process Block* defines a concurrent VHDL process which contains a series of statements to be executed sequentially. Each Process Block runs simultaneously (concurrently) during the VHDL simulation cycle.

Design Hierarchy Successive decomposition of a design entity into components, and binding of those components to other design entities that may be decomposed in a like manner, results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy*.

Model A *model* is the elaboration of the design hierarchy in the VHDL simulation environment. The model can be executed in order to simulate the design represented in the model.

3.4 Mapping SADT to VHDL

Sections 3.2 and 3.3 have provided detailed descriptions of the attributes and constructs of the SADT and VHDL subsets used in this research. These descriptions formed the foundation for defining a mapping of the SADT subset to the VHDL subset. For the purpose of this mapping definition, the symbol SADT' will be used to denote the subset of the SADT language used in this research. Similarly, VHDL' denotes the VHDL subset. Table 1 depicts the mapping definition created. The notation $X \longrightarrow Y$ is used in the table to indicate the mapping of an element of SADT' to an element of VHDL'.

Table 1. SADT' to VHDL' Mapping

SADT'	—	VHDL'
Activity	—	Design Entity
Interface Arrow	—	Interface Object (Signal)
ICOMs	—	Ports
Inputs	—	Port of mode in
Controls	—	Port of mode in
Outputs	—	Port of mode out
Mechanisms	—	Not mapped. ^a
Branches	—	Two or more unique Signals
Pipeline	—	Record Type
Cyclic Feedback	—	No Direct Mapping
Dataflow Feedback	—	Signal Feedback
Control Feedback	—	Signal Feedback
Diagrams	—	Block
Context Diagram (A-0)	—	External Block
Hierarchic Decomposition	—	Design Hierarchy
Model	—	Model

^aMechanisms were not included in the SADT subset mapped to VHDL in this research effort. See Section 7.4.2 for the explanation.

3.5 Extensions to the SADT Model

In Section 3.2.1 it was stated that SADT Activities have inputs, controls, and outputs, and that the function of the activity is to transform the inputs and controls into outputs. Sections 3.2.1 pointed out that nowhere in the diagram is there a way to describe the transformation behavior from the inputs to the outputs, or what any of the associated values might be. The Abstract Model as implemented by Tevis (48) and Kitchen (27) only allows for an informal textual description of the Activities. If detailed behavior is to be captured and later simulated, the existing SADT model must be extended.

3.5.1 SADT Activity Activation Rules In Section 3.2.1 the idea of *Activation Rules* was introduced. An SADT Activity is said to be active when it transforms its inputs and controls into outputs. Activation Rules are used to specify the combination of inputs and controls needed to generate a specific output. An activation rule consists of logic describing the preconditions which when met result in a postcondition. Marca and McGowan (32) describe the use of the ICOM codes for the activity, and the logical operators AND, OR, and NOT to generate these rules. Figure 14 illustrates an example (32:126) of activation rules for an activity titled Evaluate Job Progress (A1). The activation rules do not specify WHAT the output of the activity will be, but WHEN the particular output will be generated. In this example, the first rule $I1 \text{ and } C1 \longrightarrow O3$ states only that I1 and C1 must be present for the generation of O3 to occur. Nothing can be said about what value or information is contained in the Next Job Order Step (O3) output.

Marca and McGowan also state that generation of the activation rules for a particular activity is often difficult for the analyst. They propose the use of truth tables to generate all the possible combinations of activations for a particular activity. (32)

3.5.2 Decision Tables Working with Capt Douglass (16), it was determined that the addition of a decision table for each leaf node activity would provide a means of specifying the required detailed behavior. In the previous section, the idea of using activation rules was discussed. Recall, however, that an activation rule only allows the determination of *when* a particular activity would transform its inputs into outputs, based on the *existence*

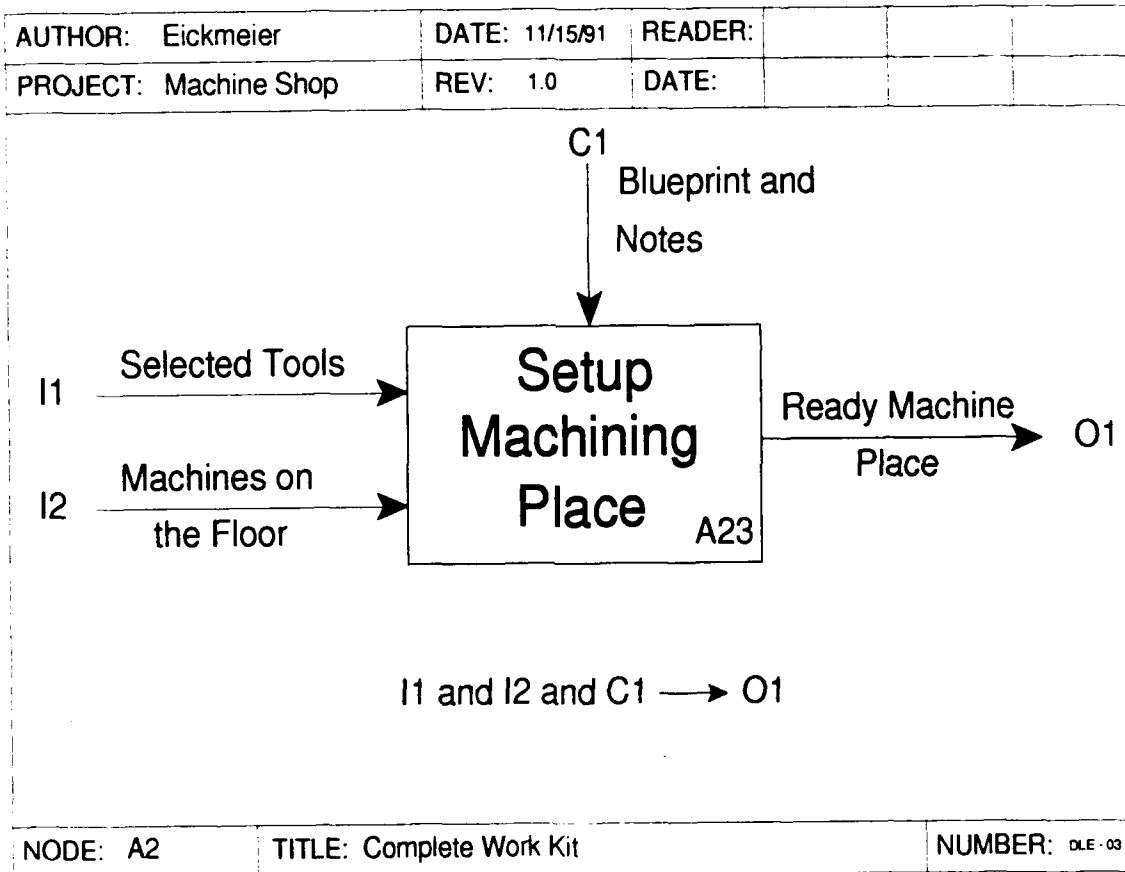


Figure 14. An Activity with an Activation Rule (32:126)

of certain inputs and controls. No mention is made of *what* values the inputs possess (pre-conditions) when they are transformed into outputs or *what* any of the associated output values are (post-conditions). Generation of an executable simulation in VHDL requires that both the *when* and *what* behavior be defined. The concept of Activation Rules is extended further here, and decision tables¹ are used to specify the state or value of all the outputs, given the state or value of all the inputs and controls.

Decision tables serve as a means to specify the detailed executable behavior necessary for the VHDL simulation, yet, the requirements analysts generating the SADT specification needs no knowledge of the VHDL syntax to generate the tables. This concept of using decision tables to specify behavior is not new. Hurle¹ (24) points out that decision tables have been used of over 30 years in software engineering.

Table 2 is an example of how the decision tables are used. This table corresponds to

Table 2. Decision Table Example - Heater Activity Behavior - A1

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
	Unsafe	True
Safe	Safe	False

the Shut Off Furnace Activity (A1) of Figure 11. The inputs, controls, and outputs (state variables) are listed across the top row. The inputs and controls form the pre-conditions and appear on the left part of the table. The outputs form the post-conditions and appear on the right part of the table. The double lines delimit the pre- and post-conditions. Each subsequent row then specifies a given state or set of states that serve as the pre-conditions, and the corresponding state changes represented as the post-conditions. Thus, a typical state-based model of formal specification is achieved. Blank entries indicate a *don't care* situation for pre-conditions, and a *no change* situation for post-conditions. Orienting the

¹Marca and McGowan (32:121) do state that they have used decision tables and finite-state machines in the past to elaborate SADT models for the specification of a telephone switch.

tables with the variables across the top makes it easy to look at an entire small table at one time. For this example (Table 2), the Combustion Sensor and Fuel Flow are Controls (pre-conditions) since they enter from the top of the A1 Activity box. Shut Down is the only output (post-condition) coming out of the A1 Activity. When the Combustion Sensor value is UNSAFE, then the Shut Down takes on the value of TRUE. This is the case regardless of the value of the Fuel Flow, since it has a blank *don't care* entry. The second row indicates that when the Fuel Flow value is UNSAFE, the Shut Down output takes on the value of TRUE also. As the third row indicates, when the values of both the Combustion Sensor and the Fuel Flow are SAFE, the Shut Down output becomes FALSE.

If tables with a large number of variables are necessary, or a large number of conditions, then the orientation of the table can be changed. Figure 15 shows a template of the modified orientation, and Tables 3 and 4 are examples of tables oriented such that the variables are listed in column 1, with the pre-conditions at the top and the post-conditions at the bottom. Subsequent columns specify the values or conditions of the variables. Thus, they serve as the rows did in the previous table. Note how easy it is to break a table ori-

SADT Activity Name					
Pre-Condition Variable Name	Pre-Condition One	Pre-Condition Two	Pre-Condition Three	Pre-Condition Four	
	Post-Condition One	Post-Condition Two	Post-Condition Three	Post-Condition Four	

Figure 15. Template of the Modified Decision Table Orientation

ented in this fashion into parts. Table 4 illustrates this, as it is just the second part of Table 3. Table 3 specifies the first three possible states and Table 4 specifies the last two. For readability, the variable names are repeated in column 1 for every part of a table. For clarity, the entire set of Input and Control variables are listed in the pre-condition (upper) section, even if only two or three have values and the rest are *don't cares*.

3.5.3 Creating and Implementing the Decision Tables While creating the decision tables, the particular orientation of the table doesn't matter. The developer can use either a horizontal or vertical approach, whichever is easier. What does matter is how the decision tables are represented in the essential model. One way the tables can be implemented is as a generic relationship and is covered in Chapter 5.

3.6 Levels of Executability of the VHDL Model

Executability of a specification can provide a powerful tool allowing the design analyst to test, debug, and further understand his design specification. Demonstrations of the specification behavior using a simulation environment allows the end-user to validate and explain the desired product. (52)

Obtaining the benefits of executable specifications (See Section 1.1) is the overall goal of this research area. The executability of a specification can be defined at several levels from Informal (Level 0) to Formal (Level 4). An informal executable specification provides insight into the structure or syntax of the system. A Formal executable specification provides visibility of the system behavior or semantics in addition to the syntax. (50)

Several levels of executability are possible in this research using the SADT and VHDL model. These levels, beginning from the lowest, informal level and progressing to the highest, formal level, are contained in Table 5. The third level of executability (Level 3) is implemented in this thesis.

3.7 Summary

This chapter has defined subsets of SADT and of VHDL, and presented a mapping from the SADT subset into the VHDL subset. This mapping serves as the basis for

Table 3. Decision Table Example – Lift Activity Behavior - A21 Part 1

Update Min/Max			
New Sched 1.FL 1 Stop			
New Sched 1.FL 1 Dest			
New Sched 1.FL 2 Stop			
New Sched 1.FL 2 Dest			
New Sched 1.FL 3 Stop			
New Sched 1.FL 3 Dest			
New Sched 1.FL 1 Up			
New Sched 1.FL 2 Up			
New Sched 1.FL 2 Down			
New Sched 1.FL 3 Down			
New Sched 1.Max			
New Sched 1.Min			
New Sched 2.FL 1 Stop			
New Sched 2.FL 1 Dest			
New Sched 2.FL 2 Stop			
New Sched 2.FL 2 Dest			
New Sched 2.FL 3 Stop			
New Sched 2.FL 3 Dest			
New Sched 2.FL 1 Up			
New Sched 2.FL 2 Up			
New Sched 2.FL 2 Down			
New Sched 2.FL 3 Down			
New Sched 2.Max			
New Sched 2.Min			
Floor	0	> New Sched 1.Max	< New Sched 1.Min
Floor	0	> New Sched 1.Min	< New Sched 1.Max
Lift	0	1	1
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	New Sched 1.Max	Floor	New Sched 1.Max
Sched 1 Mod.Min	New Sched 1.Min	New Sched 1.Min	Floor
Sched 2 Mod.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop
Sched 2 Mod.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest
Sched 2 Mod.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop
Sched 2 Mod.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest
Sched 2 Mod.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop
Sched 2 Mod.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest
Sched 2 Mod.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up
Sched 2 Mod.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up
Sched 2 Mod.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down
Sched 2 Mod.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down
Sched 2 Mod.Max	New Sched 2.Max	New Sched 2.Max	New Sched 2.Max
Sched 2 Mod.Min	New Sched 2.Max	New Sched 2.Max	New Sched 2.Max

Table 4. Decision Table Example - Lift Activity Behavior - A21 Part 2

Update Min/Max		
New Sched 1.FL 1 Stop		
New Sched 1.FL 1 Dest		
New Sched 1.FL 2 Stop		
New Sched 1.FL 2 Dest		
New Sched 1.FL 3 Stop		
New Sched 1.FL 3 Dest		
New Sched 1.FL 1 Up		
New Sched 1.FL 2 Up		
New Sched 1.FL 2 Down		
New Sched 1.FL 3 Down		
New Sched 1.Max		
New Sched 1.Min		
New Sched 2.FL 1 Stop		
New Sched 2.FL 1 Dest		
New Sched 2.FL 2 Stop		
New Sched 2.FL 2 Dest		
New Sched 2.FL 3 Stop		
New Sched 2.FL 3 Dest		
New Sched 2.FL 1 Up		
New Sched 2.FL 2 Up		
New Sched 2.FL 2 Down		
New Sched 2.FL 3 Down		
New Sched 2.Max		
New Sched 2.Min		
Floor	$\geq$ New Sched 2.Max	$\leq$ New Sched 2.Min
Floor	$>$ New Sched 2.Min	$<$ New Sched 2.Max
Lift	2	2
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	New Sched 1.Max	New Sched 1.Max
Sched 1 Mod.Min	New Sched 1.Min	New Sched 1.Min
Sched 2 Mod.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop
Sched 2 Mod.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest
Sched 2 Mod.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop
Sched 2 Mod.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest
Sched 2 Mod.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop
Sched 2 Mod.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest
Sched 2 Mod.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up
Sched 2 Mod.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up
Sched 2 Mod.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down
Sched 2 Mod.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down
Sched 2 Mod.Max	Floor	New Sched 2.Max
Sched 2 Mod.Min	New Sched 2.Min	Floor

Table 5. Potential Levels of Executability

Level 0	Obtainable with the basic SADT model. The SADT method makes provisions for describing the structure of the system under development, but does not provide a means to describe the behavior. The only semantic information allowed is the specification of the ICOMs, and the rule that Inputs and Controls are transformed into Outputs. Mapping SADT to VHDL at this level would only provide structural information, without any behavior.
Level 1	Obtainable with the SADT model augmented with Activation Rules. This would allow specification of more semantic information about <i>when</i> Inputs and Controls are transformed into Outputs, but nothing about <i>how</i> . Mapping to VHDL at this level would provide little additional information from Level 0.
Level 2	Obtainable with the SADT model augmented with decision table logic to specify the behavior of the leaf node activities. Manual mapping at the leaf node activities into a VHDL simulation provides syntax and semantic information about the SADT model, but only at the lowest level of abstraction.
Level 3	Obtainable with the SADT model augmented with decision table logic to specify the behavior of the leaf node activities. Manual mapping into VHDL of the entire hierarchal SADT model. The resulting VHDL simulation provides syntax and semantic information at each level in the SADT design. This hierarchical mapping allows for partial design completeness. This can be beneficial in a large design where subsystems (or subtrees of the SADT model) are being performed by separate design teams. Each team can test their own subsystem using a separate VHDL simulation, then the subsystems can be combined to test the entire system behavior. This level is implemented in this thesis.
Level 4	Obtainable by replacing the manual translation process of Level 3 with an automated translation of the SADT model into VHDL. A design of an automated transformation process is described in Chapter V of this thesis. The main benefit of this level is the concept of throw-away code. This concept proposed by Balzer (5, 6) emphasizes the maintenance and change of the specification instead of the resulting source code. In the context of this research, rapid changes can be made to the SADT specification of the problem with little concern for the resulting changes in the generation of the VHDL simulation. Changes can be made in response to user interaction, and the resulting specification can be simulated to demonstrate the new behavior. Thus, the benefits of rapid prototyping can be realized without regard to any implementation decisions.

developing an algorithm to transform an SADT specification into a VHDL specification for use in the validation of a system specification using simulation. This algorithm is developed and validated in Chapter IV using two test cases: 1) The Heating System and 2) The Lift Control System. Chapter IV presents the development of the SADT specification for these two problems, and the manual transformation of the Heating System specification into a VHDL specification and simulation. Chapter V then outlines a design for using the transformation algorithm to automatically generate VHDL source code.

IV. Manual Implementation of the SADT to VHDL Transformation Algorithm

4.1 Introduction

In Chapter III, a mapping was developed to transform an SADT specification into a VHDL specification. This chapter describes how this mapping was used to manually implement the transformation for two example problems: The Heating System, and The Lift Control System. The goals of this manual implementation come directly from the research objectives of this research:

1. Validate the SADT to VHDL transformation using the mapping defined in Chapter III.
2. Evaluate the benefits of simulating a specification using VHDL.

Accomplishment of these two goals is presented in this chapter by following the development of the Heating System problem first, then the Lift Control System problem.

¹ The Heating System and Lift Control System problems come from the problem set for the Fourth International Workshop on Software Specification and Design (17). This problem set contains four problem specifications: Library, Heating System, Formatter, and Lift. These problem specifications were provided in the original 'call for papers' for the workshop, and papers submitted to the workshop were to use these problems as their examples. The Heating System (17:ix) problem was based on a problem by S. White presented to the 1984 Embedded Computer System Requirement Workshop. The Lift Control System (17:x) problem is based on a problem by N. Davis of STCIDEDEC Ltd. In each case, the problem statement is given, followed by an SADT analysis and specification for the problem. Next, the generation and simulation of the VHDL source code is described. After the two problems have been described, a formal definition is given in Section 4.9 of the steps taken in generating the VHDL source.

¹These two problems were chosen by the faculty advisors of this thesis effort. The Heating System problem provided a small, manageable exercise for demonstrating initial feasibility of the SADT to VHDL transformation. The Lift Control System problem was a more complex problem which introduced the aspects of real-time behavior. This exercise served as a further test of the transformation algorithm for a larger domain of problems.

4.2 The Heating System Problem

The Heating System problem (See Section 4.1) statement specifies the requirements of a home heating system furnace controller. The furnace heats the home by circulating hot water through the house. Water is heated in this furnace using oil combustion. The complete problem statement (17:ix) for this problem was given as follows:

The controller of an oil hot water home heating system regulates in-flow of heat, by turning the furnace on and off, and monitors the status of combustion and fuel flow of the furnace system, provided the master switch is set to the HEAT position. The controller activates the furnace whenever the home [air] temperature, t , falls below $t_r - 2$ degrees, where t_r is the desired [air] temperature set by the user. The activation procedure is as follows:

- R1. the controller signals the motor to be activated.
- R2. the controller monitors the motor speed and once the speed is adequate it signals the ignition and oil valve to be activated.
- R3. the controller monitors the water temperature and once the temperature has reached a predefined value it signals the circulation valve to be opened. The heated water then starts to circulate through the house.
- R4. a fuel flow indicator and an optical combustion sensor signal the controller if abnormalities occur. In this case the controller signals the system to be shut off.
- R5. once the home temperature reaches $t_r + 2$ degrees, the controller deactivates the furnace by first closing the oil valve and then, after 5 seconds, stopping the motor.

In addition the system is subject to the following constraints:

- C1. minimum time for furnace restart after prior operation is 5 minutes.
- C2. furnace turn-off shall be indicated within 5 seconds of master switch shut off or fuel flow shut off.

4.2.1 SADT Analysis The Heating System was analyzed using SADT and is presented in Figures 16 to 19. Figure 16 shows the Heating System Environment. This diagram represents the A-0 level or Context Diagram for the Heating System. The Heating System SADT model has two time-based constraints: Five Sec Timer and Five Min Timer which correspond to the timing constraints of the problem statement. The generation of these timing constraints was considered external to the Heating System. Treating

the timing as external events in this way allows them to be viewed as control events on the A-0 (Context) diagram. This viewpoint separates the functional behavior of the system from the time-related behavior and is similar to Booch's approach to the Heating problem in his object oriented design textbook (11). Separating the time-related behavior allowed a common model to be used in this research and that of Douglass (16). The SADT model and corresponding VHDL simulation is expanded in Section 4.4 to address the simulation of the time-based events.

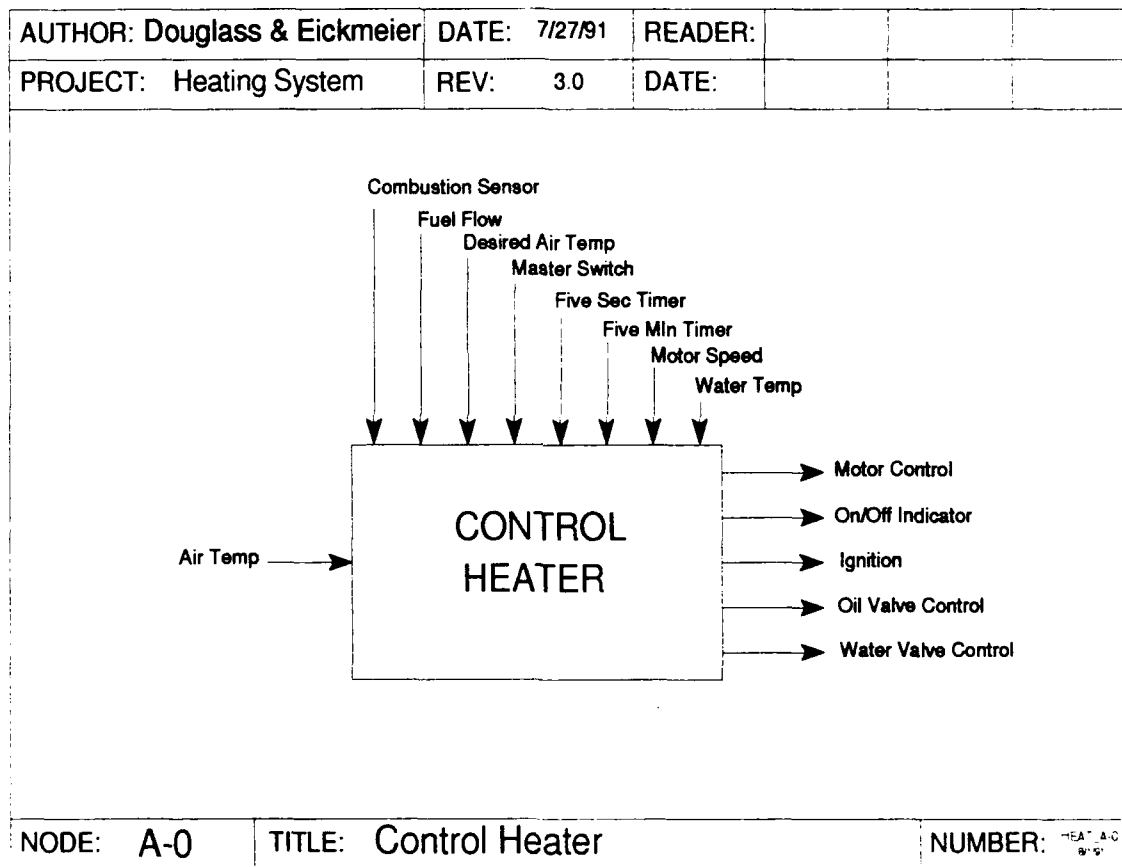


Figure 16. Heating System A-0 Diagram

In addition to the time-based constraints, Figure 16 shows the remaining external interfaces. The Air Temperature is the only external input to the system. The other external drivers to the system (Master Switch, Fuel Flow, etc.) are controls that help determine when to initiate the Heater Control system. It's important to note here that an

assumption was made that the controller being analyzed is the software for the controller. Hardware devices such as the motor, valves, ignition, and sensors are external to the system.

Figure 17 represents a first level decomposition of the A-0 diagram shown in Figure 16. The Shut Off Furnace activity (A1) monitors the Combustion Sensor (C1) and Fuel Flow indicator (C2), and acts to shut down the system for any abnormal conditions by generating the Shut Down output. This Shut down signal is provided to the Control

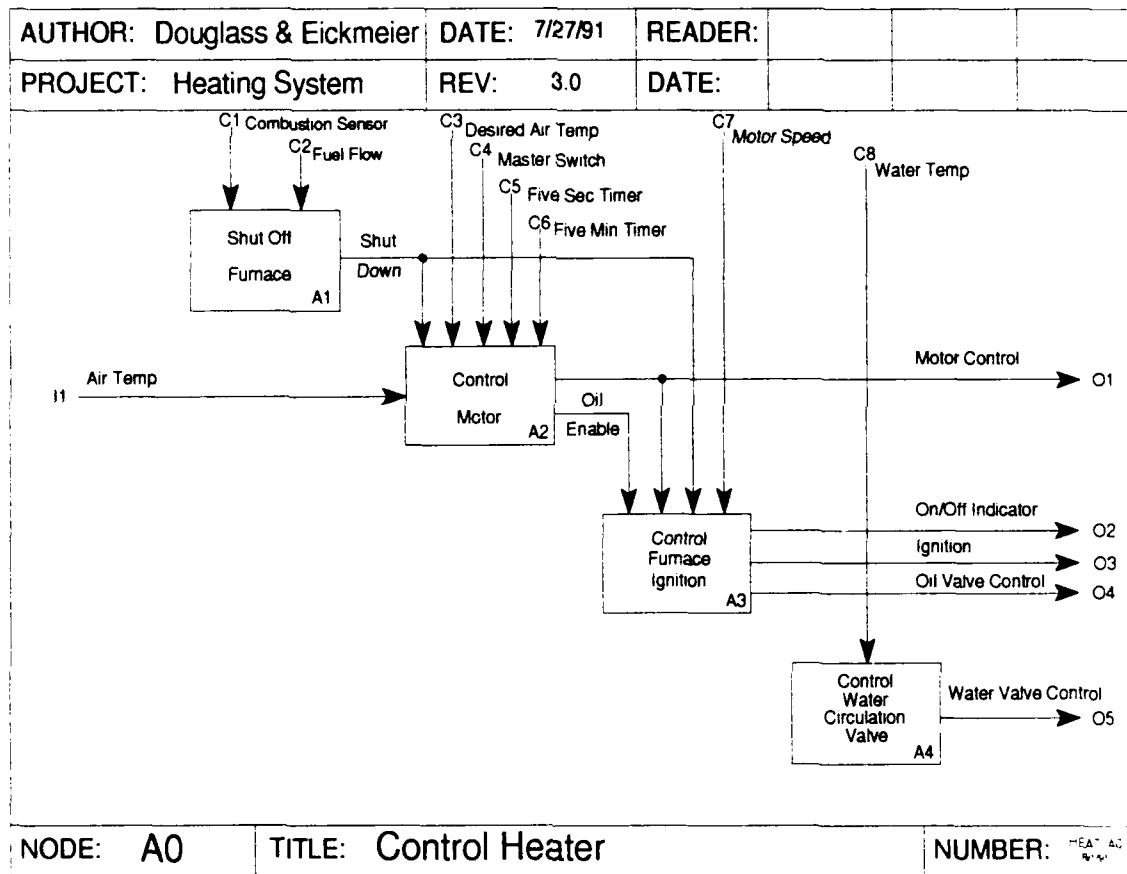


Figure 17. Heating System A0 Diagram

Motor (A2) and Control Furnace Ignition (A3) activities, so those activities can process an orderly shut down. The Shut Down signal is not provided to the Control Water Circulation Valve activity (A4), since the sole control for this activity is the Water Temp (C8). The assumption was made that this is a desirable safety feature to prevent excessive water

pressure build-up (this and other assumptions concerning the operation of the Heater Controller are described in Section 4.2.2). The pre-defined temperature to activate the water valve is not considered as an input. Instead, it is just an internal setting. The Control Motor activity (A2) receives the Air Temperature input (I1) and the Desired Air Temp control (C3) to determine when to activate both the external Motor Control (O1), and the Oil enable, which is a control provided to the Control Furnace Ignition activity (A3). The Master Switch (C4) is also a control to the Control Motor activity (A2), and provides for normal operation or an orderly shut down of the system. The Control Furnace Ignition activity (A3) receives the Oil enable and Motor Control (O1) outputs from the Control Motor activity (A2) along with the external Motor Speed control (C7). The Motor Control (O1) is passed to the Control Furnace Ignition activity (A3) to indicate when the system is running, so the On/Off indicator (O2) can be controlled. This activity provides the On/Off indicator output (O2), along with the Ignition (O3) and Oil Valve Control (O4) outputs.

The A0 level diagram (Figure 17) is further decomposed in the A2 (Figure 18), and A3 (Figure 19) diagrams. The Shut Off Furnace (A1) and Control Water Circulation Valve (A4) activities are not decomposed any further. In other words, they are leaf node activities. The breakdown of the Control Motor (A2 of Figure 17) activity is presented in Figure 18. The Compare Temperature activity (A21) is the heart of this breakdown, receiving the Air Temperature input (I1), along with the Desired Air Temp control (C2) to determine when the furnace should activate. The Master Switch (C3) and Shut down (C1) controls are also provided to this activity to control shut down activities. The output of the Compare Temperature activity (A21) is Oil enable (O2), which feeds the Control Furnace Ignition activity (A3) of Figure 17 and the Activate Motor activity (A22 of Figure 18). Once the Oil enable (O2) is activated, and the Five Minute Timer event (C5) is true, Motor Control (O1) is turned on by the Activate Motor activity (A22). This ensures that the furnace does not turn on until at least five minutes has expired since the last running. Once the Oil enable (O2) is deactivated, the Activate Motor (A22) waits for the Five Second Timer (C4) to become true before deactivating the Motor Control output (O1). This ensures that the motor will continue to run for five seconds after the oil has been

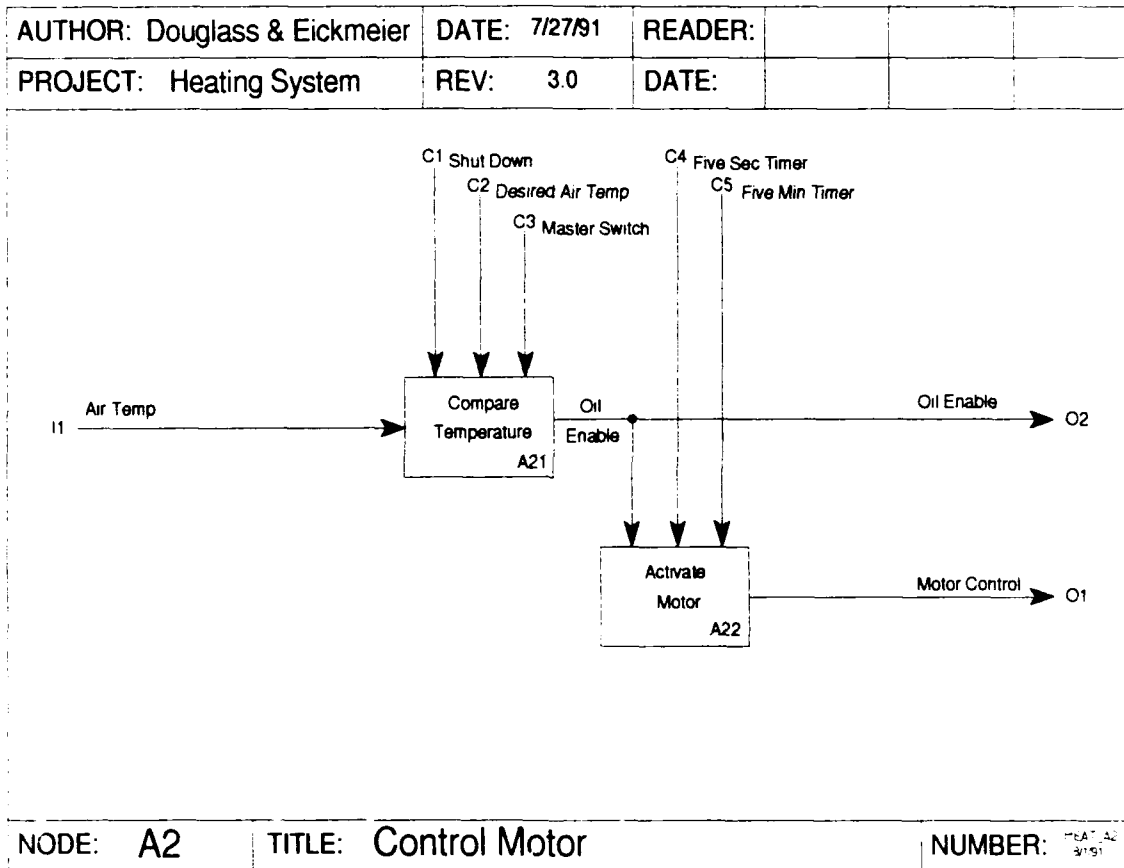


Figure 18. Heating System A2 Diagram

shut off under normal shut down conditions. Any abnormal shut off condition, caused by either the Master Switch (C3) or the shut down (C3) control, is handled by deactivating the Oil enable (O2). This in turn deactivates the Activate Motor activity (A22) and also deactivates the Activate Oil Valve activity (A33) on Figure 19. By deactivating the Activate Motor (A22), the external Motor Control output (O1) is also turned off.

Figure 19 presents the final breakdown diagram of the Control Furnace Ignition activity (A3 of Figure 17). The Monitor Motor Speed activity (A31) receives the Motor Control (C2) and Motor Speed (C4) controls. The Motor Control (C2) is used to set the On/Off indicator (O1). When the Motor Control (C2) is activated, the Motor Speed (C4) is monitored for the desired speed. When these conditions are valid, the Desired Speed output is activated. This output is fed to the Activate Ignition activity (A32) which then provides the external Ignition output (O2). The Desired Speed is also fed to the Activate Oil Valve activity (A33). When Desired Speed is activated, along with the Oil enable (C1), the Activate Oil Valve activity (A33) turns the external Oil Valve Control output (O3) on. Shut down (C3) is brought in to both the Activate Ignition (A32) and Activate Oil Valve (A33) activities to provide for emergency shut off.

4.2.2 Problems Encountered Several problems were encountered with the given problem statement which forced some assumptions to be made.

- Inconsistencies - conflicting and ambiguous items in the specification:
 - Fuel flow shut off terminology (Constraint 2) versus oil valve shut off (Requirement 5). The assumption was made that fuel flow shut off meant a normal shut down condition, such as when the desired temperature is reached.
- Incomplete Specification:
 1. What steps to take when the Master Switch is not on the HEAT position.
 2. Exactly what steps to take when an error based Shut-Down occurs.
 3. When to shut off the Water Valve.
 4. When is the ignition turned off.

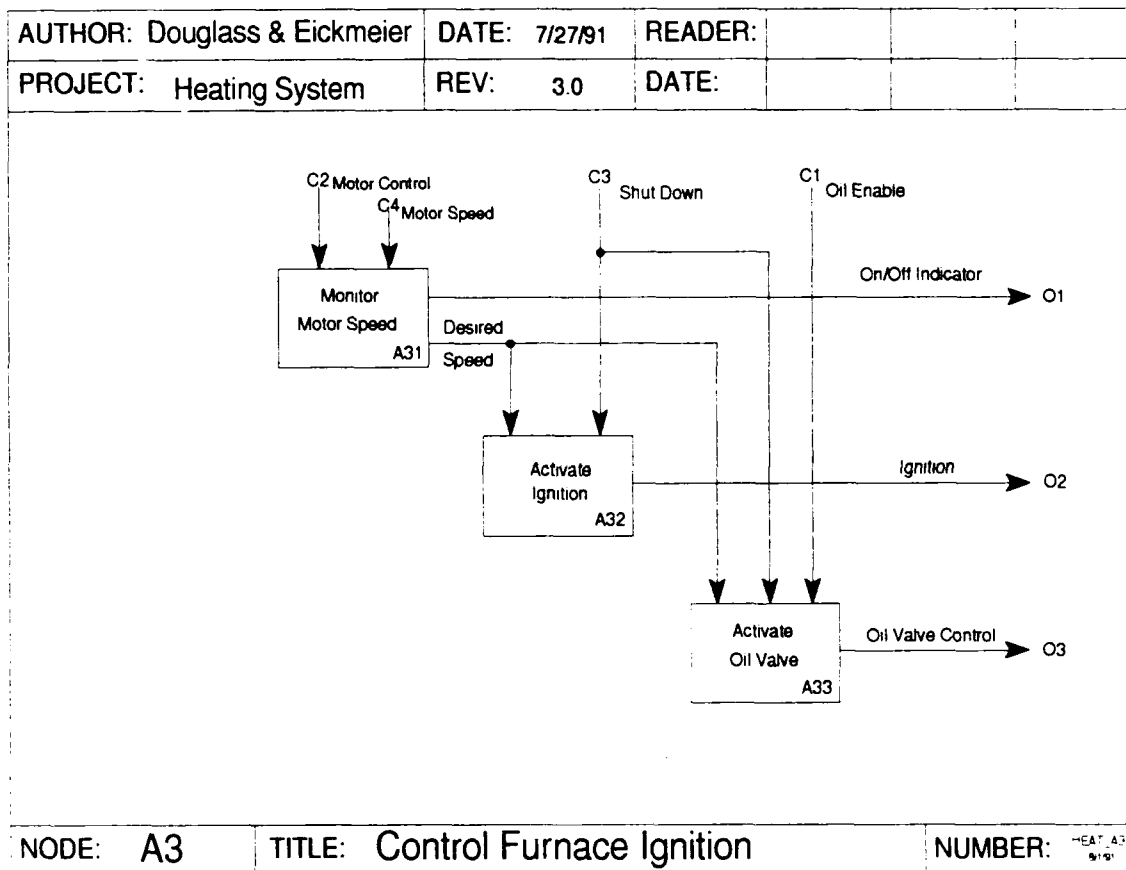


Figure 19. Heating System A3 Diagram

5. when Exactly is the On/Off indicator turned on.

The following assumptions were made to overcome these incomplete items:

1. Assume that if the Master Switch is not in the HEAT position, then shut down the system at the Compare Temperature activity.
2. Turn everything off immediately EXCEPT the Water Valve.
3. Shut off the water valve only after the water temperature goes below the previously defined threshold.
4. The ignition is turned off with the motor, 5 seconds after the oil valve is closed. This allows any excess oil to burn off.
5. The On/Off indicator is turned on when the motor control is activated.

4.2.3 Decision Tables for the Leaf Node Activities Decision Tables were used (See Section 3.5 for justification) to specify the behavior of the leaf node (lowest level) activities and are provided in Tables 18-24 in Appendix A. Table 6 shows an example of a decision table for the Heating System Problem. The reader should note that the inputs and controls are listed across the top of the tables followed by the outputs. Each row represents a particular condition or state of the system, along with the required output(s) for that condition or state.

Table 6. Decision Table Example – Heater Activity Behavior - A1

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
	Unsafe	True
Safe	Safe	False

4.2.4 Test Cases The goal of this research effort is the development and validation of an algorithm which maps SADT specifications into VHDL specifications. In keeping with

the goals of prototyping of behavior (See Section 2.2), exhaustive testing of the Heating System specification was not the goal of the following test cases. Instead, four test cases were developed to test the basic functional behavior of the specified Heating System. In the test case description, the requirement tested is indicated. The requirements listed in the problem statement under the "activation procedure" section are referenced as *R1* through *R5*. The requirements listed under the "constraints" section are referenced as *C1* and *C2*. Table 25 in Appendix A is a Key for the data items and their applicable states. Note that a Δt is used to represent the difference between the measured air temperature and the desired air temperature.

1. TEST 1 - This tests the system from an initial startup condition. There are three phases to this test, the first for the condition where the temperature differential between the desired temperature and the actual room temperature is 2 degrees or more. The second phase occurs after the motor reaches the desired speed and the third phase for when the water temperature has reached the pre-defined level. See Appendix A, Tables 26-28 for the expected output results. Requirements Tested: *R1*, *R2*, and *R3*.
2. TEST 2 - This tests the system for a normal shut off. There are also three phases for this test. The first is for right after the temperature differential between the desired temperature and the actual temperature is minus 2 degrees or more. The second phase is for when the motor speed drops below the threshold, and the third is for when the water temperature drops below the threshold. See Appendix A, Tables 29-31 for the expected output results. Requirement Tested: *R5*.
3. TEST 3 - This test case covers error condition shut downs and has two phases. The first is for a Combustion error and the second is for a fuel flow error. See Appendix A, Tables 32 and 33 for the expected output results. Requirement Tested: *R4*.
4. TEST 4 - Test Case 4 covers a shut down resulting from turning the Master Switch off of the HEAT position and has three phases. The first is before 5 seconds has expired, and the second is for after 5 seconds (the motor should continue to run for 5 seconds after the oil and ignition get shut off). The third phase is for when the water temperature has fallen below the threshold, and by this phase all outputs should either be off or false. See Appendix A, Tables 34-36 for the expected output results. Requirement Tested: *R5*.
5. TEST 5 - In TEST 5, the system constraint *C1* was tested. Constraint *C1* states that 5 minutes must expire after shut down before the heater may start again. In this test, TEST 1 (normal start up) is executed 1 minute after the shut down of TEST 4. At this point, the system is not expected to re-start. An additional four minutes is allowed to expire, and the system is expected to be in the same state as Phase 3

of TEST 1. Execution of this step required the version of the Heating System which added timing behavior (See Section 4.4).

4.3 VHDL Source Code Generation for the Heating System

Manual transformation of the Heating System SADT specification into executable VHDL code was the next phase in validating the SADT to VHDL mapping presented in Chapter III. During the manual transformation, a step-by-step algorithm for generating the VHDL source code from an SADT specification was developed. This algorithm is presented in this chapter.

The following discussion will first proceed informally through the steps, giving examples of the generated source code. Section 4.9 will then formally summarize the steps in the process. The Heating System Problem will be used as the example transformation in this discussion, and the reader is referred to the complete VHDL source code for this problem in Appendix C.

4.3.1 Types Definition Package The first step in creating the VHDL source code was the generation of a types package for the SADT model. A VHDL package is similar to an Ada package; and it provides a place to define the type information which can then be used by the entire VHDL model. VHDL is a strongly typed language, so the generation of this package is critical to the generation of the entire VHDL model.

The basic definition of SADT does not contain the concept of types for interface arrows, however the AFIT IDEF₀ Extension (21) adds a data dictionary to the model. This data dictionary, and its associated entries is described by Hartrum (21) and extended by Kitchen (27). A subset of Kitchen's definition of the data dictionary entry for an interface arrow (called a Data Element in the AFIT IDEF₀ Extension) is illustrated in Figure 20.

The information contained in the data dictionary entry for an SADT interface arrow was used to create the type definition for the corresponding VHDL Interface Object. For the Heating System problem, a formal data dictionary was not developed, however Table 7 was generated as a key to the Test Cases in Appendix A. This table contained the necessary

NAME:	(data element name)
DESCRIPTION:	(text description)
DATA TYPE:	(the type of data, if known)
MIN VALUE:	(minimum data value, if known)
MAX VALUE:	(maximum data value, if known)
RANGE:	(range of values, if applicable)
VALUES:	(enumeration values, if appropriate)

Figure 20. Data Dictionary Entry Format for a Data Element (27:55)

information to generate the VHDL type information. An example data dictionary entry

Table 7. Applicable Values of Interface Arrows

Item	Applicable States	
Δt^a	≤ -2	≥ 2
Combustion Sensor	Safe	Unsafe
Fuel Flow	Safe	Unsafe
Master Switch	Heat	Off
Five Sec Timer	True	False
Five Min Timer	True	False
Motor Speed	Threshold	Below Threshold
Water Temp	Threshold	Below Threshold
Motor Control	On	Off
On/Off Indicator	On	Off
Ignition	On	Off
Oil Valve Control	Open	Closed
Water Valve Control	Open	Closed

^a Δt = Air Temp - Desired Air Temp

for the Combustion_Sensor data element from Figure 16 is shown in Figure 21. The corresponding

VHDL type definition for the Combustion_Sensor is:

```
type Control_Sensor_Type is (SAFE, UNSAFE);
```

The convention used in this research was to append the string `‘_Type’` to the data element

NAME:	Combustion_Sensor
DESCRIPTION:	External sensor indication.
DATA TYPE:	Enumerated
MIN VALUE:	
MAX VALUE:	
RANGE:	
VALUES:	SAFE, UNSAFE

Figure 21. Data Dictionary Entry for Combustion_Sensor Arrow

name when generating the type declaration. This allowed for the actual data element name to be used when it was later instantiated, for example:

```
Combustion_Sensor    :    in  Combustion_Sensor_Type;
```

Similar definitions were created for each SADT interface arrow. The complete type definition package for the Heating System is in Appendix C, Section C.1.

4.3.2 Entity Declarations Once the types package had been generated, the next step was to create the basic VHDL building blocks, the design entities. To use a design entity two things must be generated: 1) an entity declaration, and 2) an architectural body. This section describes the creation of entity declarations for each activity in the SADT model. The generation of the architectural bodies will be discussed in Sections 4.3.3 and 4.3.4.

As defined in Section 3.3.1, an entity declaration specifies the interfaces of the design entity to the external environment. This correlates to the Inputs, Controls, and Outputs of the SADT activity. As an example, consider the A21 activity, Compare Temperature, shown on Figure 18. This activity interfaces with its surrounding environment via its one Input, three Controls, and one Output. The corresponding design entity has the same interfaces called ports, and would be declared as shown in Figure 22:

As specified in the mapping definition (Section 3.4), the Inputs and Controls are declared as ports of mode **in**, and the Output declared as a port of mode **out**. This same

```

entity Compare_Temperature is                                -- A21
    port ( Air_Temp      : in  Integer;                      -- I1
           Shut_Down     : in  Boolean;                      -- C1
           Des_Air_Temp  : in  Integer;                      -- C2
           Master_Switch : in  Master_Switch_Type;          -- C3
           Oil_Enable    : out Boolean);                     -- O2
end Compare_Temperature;

```

Note: The text from the "--" to the end of the line is considered a comment in VHDL.

Figure 22. Compare_Temperature Entity Declaration

process is used to generate entity declarations for each and every activity in the Heating System SADT definition. The declarations were grouped in separate files corresponding to the diagram where they originated. The VHDL source code for the entity declarations are contained in Appendix C, Sections C.3, C.5, C.8, and C.10. The reader should note the statement

```
use Work.Heater_Package.all;
```

proceeds each entity declaration. This allows the entity declaration to 'use' the type definitions package which was created in the first step of code generation (Section 4.3.1). For the Heating System, the types package was named *Heater.Package* and the VHDL library name was *Work*.

4.3.3 Architecture Body Declarations - Behavioral For each entity in the VHDL model, an architectural body must be generated. The architecture body can be of two basic forms: behavioral and structural. The generation of the behavioral architecture descriptions is discussed first. The generation proceeded using a 'bottom-up' approach starting with entities corresponding to the leaf-node activities of the SADT model.

For the VHDL subset used in this research, a behavioral architecture body contains three separate sections:

1. Declaration Statement

2. Process Block

3. Sequential Statements representing the behavior of the entity.

An architecture body begins with a declaration statement which names the unit. The architecture declaration for activity A1, Shut Off Furnace (contained in Figure 17), looks like:

```
architecture Behavior of Shut_Off_Furnace is
begin
    {Statements}
end Behavior;
```

Recall from Section 3.5 that the extended SADT model captures the detailed behavior of the model at the leaf-node activities using decision tables. This behavior needs to be represented in the architecture body declaration. The VHDL *Process Block* was chosen to contain this detailed behavior description. A *Process Block* defines a concurrent VHDL process which contains a series of statements to be executed sequentially. Each Process Block runs simultaneously (concurrently) during the VHDL simulation cycle. In Section 3.2.1 the SADT Activity was also described as a concurrent process. Thus, the VHDL Process Block models this behavior. Recall, however, that the SADT Activity was constrained by its Inputs and Controls, i.e. their presence was required before the Activity could proceed. This same behavior can be duplicated in the VHDL Process Block by adding a *Sensitivity List* to the Process definition. This Sensitivity List can contain one or more signals which must be present prior to the execution of the block of sequential statements. As an example, the next part of the architecture body for the Shut Off Furnace Activity can be generated as shown in Figure 23.

The process block contained in the architecture body would only execute when an *event* occurs on either the Combustion_Sensor or Fuel_Flow signals. "An *event* is said to occur on a signal when the current value of the signal changes" (25:B-5).

Now that a block structure has been chosen to represent the constrained behavior of the SADT Activity box. A VHDL representation of *WHEN* the Activity transforms


```

architecture Behavior of Shut_Off_Furnace is
begin
    process (Combustion_Sensor, Fuel_Flow) begin
        {Statements}
    end process;
end Behavior;

```

Figure 23. Process Block Declaration for Shut_Off_Furnace

its Inputs and Controls into Outputs, and *WHAT* value of the Output will be generated. The discussion on Activation Rules in 3.5.1 stated that the decision tables created for the leaf-node activities define this behavior.

Table 8 shows an example decision table from the Heating System problem. As de-

Table 8. Decision Table Example – Heater Activity Behavior - A1

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
	Unsafe	True
Safe	Safe	False

scribed in Section 3.5 the first two columns represent the pre-condition(s) necessary for the Shut Off Furnace Activity to generate the post-condition contained in the third column. Each row contains a set of pre and post conditions. This behavior maps directly to a VHDL **if-then-elsif-end if** construct². The first pre-condition/post-condition pair forms the initial **if {condition} then {statement}**. Each subsequent pair forms an **elsif {condition} then {statement}**. Since the decision table contains condition pairs for every transformation of interest, no final **else {statement}** is needed.

This provides the additional information needed to complete the architecture body for the Shut Off Furnace activity. The complete definition is provided in Figure 24.

²An alternative use of a VHDL **if-then-end if** construct is discussed in Section 6.2.1.2

```

architecture Behavior of Shut_Off_Furnace is
begin
    process (Combustion_Sensor, Fuel_Flow) begin
        if Combustion_Sensor = UNSAFE then
            Shut_Down <= TRUE;
        elsif Fuel_Flow = UNSAFE then
            Shut_Down <= TRUE;
        elsif Combustion_Sensor = SAFE and Fuel_Flow = SAFE then
            Shut_Down <= FALSE;
        end if;
    end process;
end Behavior;

```

Figure 24. Complete Architecture Body Definition for Shut_Off_Furnace

4.3.4 Architecture Body Declarations - Structural After the behavior of the model is described in the behavioral architecture bodies, the hierarchical structure of the SADT model needs to be built. This hierarchy is defined in the structural form of an architecture body. For the SADT to VHDL transformation process, the architecture structure definition has four distinct sections:

1. Declaration Statement
2. Component Declarations
3. Local Signal Declarations
4. Component Instantiations and Signal Assignments

The declaration statement is similar to the behavioral declaration statement described in the last section. The Control Motor (A2) activity (Figure 17) will be used as the example for this section. An equivalent diagram showing the Control Motor entity in a more hardware component representation is provided in Figure 25. The complete VHDL architecture body for this activity is contained in Appendix C, Section C.6. The Control Motor activity is a parent activity (non leaf-node), and it is defined in the SADT model using decomposition. In this case, Control Motor was decomposed into two child activities: Compare Temperature (A21) and Activate Motor (A22) shown on Figure 18.

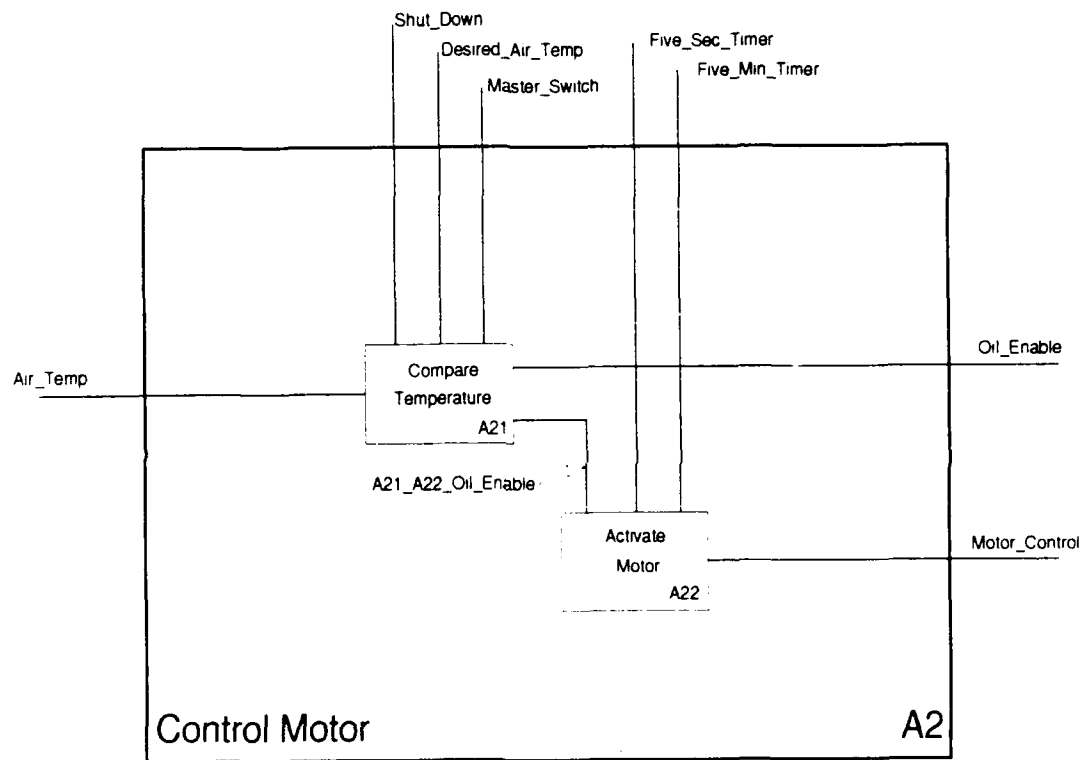


Figure 25. Component Representation for Control_Motor

The same concept of decomposition is shown in Figure 25. To begin the architecture body, the template shown in Figure 26 is generated.

```
architecture Structure of Control_Motor is
    {Component Declarations}
    {Signal Declarations}
begin
    {Component Instantiations}
    {Signal Assignments}
end structure;
```

Figure 26. Architecture Body for Control_Motor

The first section of the structural body is the component declarations. This section declares what other design entities are used in the decomposition of the entity being defined. In other words, a component declaration is required for each child of the entity being defined. In our Control Motor example, the entities Compare Temperature (A21) and Activate Motor (A22) are used in the decomposition. Forming the component declaration follows the same process used to generate an entity declaration as described in Section 4.3.2. The component declarations for Compare Temperature and Activate Motor are shown in Figure 27.

The second section of the architecture body contains declarations of local signals needed in the final, component instantiation, section. Section 3.3.1 defined an *Interface Object* as the channel of communication between design units. The Interface Objects used here are the signals and the ports (both defined in Section 3.3.1). Signals are the VHDL component corresponding to the SADT interface arrow. Local signals need to be declared for each non-boundary arrow on the child diagram. Recall from Section 3.2.2, a boundary arrow is an arrow which originates on the parent diagram, and is an external input or output on the child diagram. On Figure 18, the only non-boundary arrow is the branch (defined in Section 3.2.3) of the Oil Enable arrow which connects Compare Temperature (A21) and Activate Motor (A22). Note that the other branch of Oil Enable (O2) is a boundary arrow. A definition in VHDL terminology would be: a local signal needs to be declared for every interface between two or more components declared in the Component

```

component Compare_Temperature
port ( Air_Temp      : in Integer;
      Shut_Down     : in Boolean;
      Des_Air_Temp  : in Integer;
      Master_Switch : in Master_Switch_Type;
      Oil_Enable     : out Boolean);
end component;

component Activate_Motor
port ( Oil_Enable     : in Boolean;
      Five_Sec_Timer : in Boolean;
      Five_Min_Timer : in Boolean;
      Motor_Control   : out Motor_Control_Type );
end component;

```

Figure 27. Component Declarations for Control_Motor

Declaration section. Figure 25 shows a separate connection between A21 and A22 to comply with this requirement. The signal declaration for the Control Motor architecture looks like

```

signal A21_A22_Oil_Enable : Boolean;

```

The signal name can be any string of characters, but the naming convention chosen here identifies the SADT activities being connected. This signal will be used in the port map portion of the component instantiations for Compare Temperature (A21) and Activate Motor (A22).

The final section of the architecture definition is the Component Instantiations and Signal Assignments. This section specifies actual Component names which will be referenced in the VHDL simulation. A *Port Map* is associated with the component instantiation and logically describes how the components are connected to each other on the parent diagram. As an example, the component instantiations from the Control Motor architecture are given in Figure 28. In this instantiation, the external inputs Air_Temp, Shut_Down, Des_Air_Temp, and Master_Switch are connected to the first four **in** ports on component A21. The local signal A21_A22_Oil_Enable is connected to the A21 **out** port. This local signal is then connected to the first **in** port on component A22. The remaining connections

on A22 are the external inputs Five_Sec_Timer, Five_Min_Timer, and the external output Motor_Control respectively.

```
A21: Compare_Temperature port map (  
    Air_Temp,           -- in  
    Shut_Down,         -- in  
    Des_Air_Temp,      -- in  
    Master_Switch,     -- in  
    A21_A22_Oil_Enable ); -- out  
  
A22: Activate_Motor port map (  
    A21_A22_Oil_Enable, -- in  
    Five_Sec_Timer,    -- in  
    Five_Min_Timer,    -- in  
    Motor_Control );   -- out
```

Figure 28. Component Instantiations for Control_Motor

The position of the signal (recall from Section 3.3.1 that ports are instances of signals) in the port map define the connections to that port. This is defined as *positional association* in VHDL terminology.

Finally, signal assignments may be necessary in the architecture body to assign values to the Output boundary arrows. One such example exists in the Control Motor structure of Figure 18. A local signal was declared and used in the port maps to connect A21 and A22 together. Thus, the value of the Oil Enable output from A21 was transferred to A22 via that local signal. One should note, however, that Oil Enable is also a boundary arrow (ICOM-O2) which returns a value to the parent diagram. This arrow (VHDL signal) does NOT get the value output from A21 unless explicitly assigned. Therefore, the following statement is needed after the component instantiations:

```
Oil_Enable <= A21_A22_Oil_Enable;
```

This completes the definition of all the sections of the architecture body for the Control Motor Activity. Combining all the sections yields the source code shown in Figure 29.

```

architecture Structure of Control_Motor is
    component Compare_Temperature
    port ( Air_Temp      : in Integer;
          Shut_Down    : in Boolean;
          Des_Air_Temp  : in Integer;
          Master_Switch : in Master_Switch_Type;
          Oil_Enable    : out Boolean);
    end component;

    component Activate_Motor
    port ( Oil_Enable    : in Boolean;
          Five_Sec_Timer : in Boolean;
          Five_Min_Timer : in Boolean;
          Motor_Control  : out Motor_Control_Type );
    end component;

--SIGNAL DECLARATIONS

    signal A21_A22_Oil_Enable : Boolean;

begin
    A21: Compare_Temperature port map (
        Air_Temp,
        Shut_Down,
        Des_Air_Temp,
        Master_Switch,
        A21_A22_Oil_Enable );

    A22: Activate_Motor port map (
        A21_A22_Oil_Enable,
        Five_Sec_Timer,
        Five_Min_Timer,
        Motor_Control );

    Oil_Enable <= A21_A22_Oil_Enable;
end structure;

```

Figure 29. Complete Architecture Definition for Control_Motor

The process of generating the structural architecture definitions continues for every SADT activity which has child activities. The tree structure diagram from Chapter III is redrawn here in Figure 30. This tree represents the decomposition of the Heating System

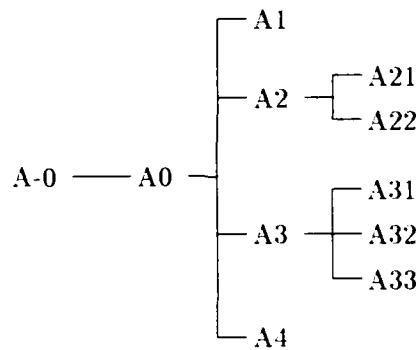


Figure 30. Heating System Tree Structure

in the SADT model previously shown in Figures 16 to 19. Using this representation, it is better understood when the definition of VHDL structure is finished. The process is complete when the entire SADT model is defined from the highest A-0 Context Diagram (the root node of the tree) down to the leaf nodes of each branch stemming from A-0. Specifically, structural definitions are required for the A-0, A0, A2, and A3 nodes.

4.4 Extending the Heating System to Address Timing

The problem statement for the Heating System contains two specific timing constraints on the behavior of the Heater Controller. These are restated as:

1. The heater may restart only after 5 minutes has elapsed from a prior operation.
2. The motor is to be stopped 5 seconds after the oil valve is closed.

The reader may have noticed what appears to be a third timing constraint in the specification statement, "furnace turn-off shall be indicated within 5 seconds of master switch shut off or fuel flow shut off." This statement indicates a response time constraint, and not a specified time the heater system must wait before the indication can occur. This response constraint is handled by generating the required indication at exactly the same

time as the shut off condition occurs. Validating that the resulting system meets such response time constraints can be performed during the VHDL simulation.

The two timing constraints are treated in the basic SADT model of the Heater System as externally generated controls; namely, Five Sec Timer and Five Min Timer. In Douglass's Refine Simulation (16) of the Heater System, these controls are assigned values as part of the test case generation. The input values for these controls are specified as part of the test inputs given in Appendix A, Tables 26-36. This same approach was used in the initial VHDL simulation of the Heater System. Once the behavior of the Heater was validated in this manner, the VHDL simulation for the Heater was extended to model the generation of the time-based controls.

Figure 31, the Control Heater Environment (A-1) diagram, represents an environment level view of the Control Heater System. This diagram is one level higher in the hierarchy than the Context (A-0) diagram (Figure 16) for the Heater System. The A-1 diagram shows two additional activities added to the SADT model to represent the sources of the time-based constraints.

The behavior of the two activities, Delay Five Min and Delay Five Sec, is dependent specifically on the way VHDL deals with simulation events. Extending an SADT model to deal with timing constraints is not something which can easily be specified in a Decision Table, and therefore, it can not be generated automatically in the SADT to VHDL transformation process. The premise of this research is that the non-time related behavior of a system is addressed in the SADT model and the corresponding Decision Tables. This model can then be translated into VHDL source code and simulated. This allows for validation of the non-time related behavior first. Once this is finished, the model can be extended to address timing by a user who understands the problem domain and the time related constructs of VHDL.

The VHDL code generated for the Delay Five Min activity is shown in Figure 32. Generation of the entity declaration is basically the same as described in Section 4.3.2 with two additional concepts introduced. First, a generic constant, Five_Min_Delay, is declared of type Time and assigned an initial value of 5 min. The use of a generic allows

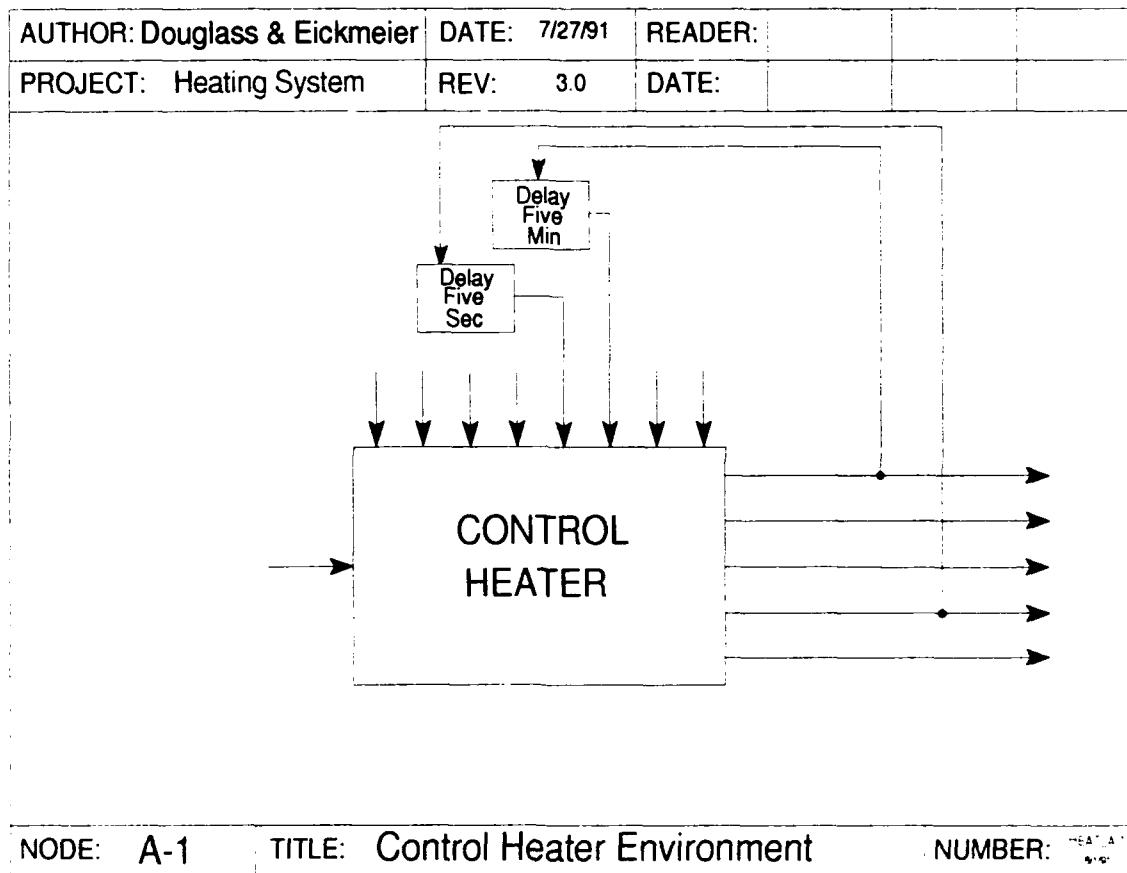


Figure 31. Heating System A-1 Diagram

```

entity Delay_Five_Min is
    generic (Five_Min_Delay : Time := 5 min);

    port    (Motor_Control  : in Motor_Control_Type;
             Five_Min_Timer : out Boolean := TRUE);
end Delay_Five_Min;

architecture Behavior of Delay_Five_Min is
begin
    process (Motor_Control) begin

        if (not Motor_Control'Stable) and
           (Motor_Control = Off_State) then
            Five_Min_Timer <= False after 0 ns,
                               True after Five_Min_Delay;
        elsif (not Motor_Control'Stable) and
              (Motor_Control = On_State) then
            Five_Min_Timer <= False;
        end if;
    end process;
end Behavior;

```

Figure 32. VHDL Code for the Delay_Five_Min_Activity

this constant to be redefined to have some value other than 5 min at a some higher level in the design architecture. The second difference shown in the port declaration is where an initial value for Five_Min_Timer is specified. The need for initial values of inputs and controls became extremely important in the code generation of the Lift Control System, and the discussion of initial values is deferred to Section 4.8. The architecture body for Delay_Five_Min employs specific VHDL constructs specific to time-based interface objects. The reader is referred to the VHDL language reference manual (25) or Lipsett, Schaefer, and Ussery (31) for definition of these constructs.

4.5 Generating the VHDL Test Environment

Following the process described in Sections 4.3 and 4.4 resulted in a completely defined SADT model. However, a test environment was needed to exercise the behavior of the model. Creating the test environment entailed creating a Testbench, a structural description of the entire system under test, and a Component Configuration declaration.

4.5.1 System Testbench In the VHDL environment, testing is performed by creating a *Testbench*. This Testbench consists of a separate VHDL entity which exercises the SADT model under test. This Testbench is connected to the SADT model (See Figure 33), and it generates the inputs, and reads the outputs of the model. The behavior of the Testbench is written by a user who is familiar with the domain of the problem and VHDL. The contents of the Testbench behavior can vary greatly, but essentially it assigns values to the SADT model inputs, and then allows the simulation to run a specified amount of time. The outputs of the SADT model are then compared to the expected results. The Testbench for the Heating System Problem is contained in Appendix C, Section C.14.

4.5.2 System Structural Declaration At this point four top level entities exist: the Control Heater entity, the two timer entities and the Testbench entity. In testing terminology, the Control Heater entity is the *Unit Under Test* or UUT. These entities now need to be connected together so testing can be performed on the UUT. A null entity (has no ports) called System is declared and a structural architecture is declared which connects the components together. This System entity will be the entity simulated in the

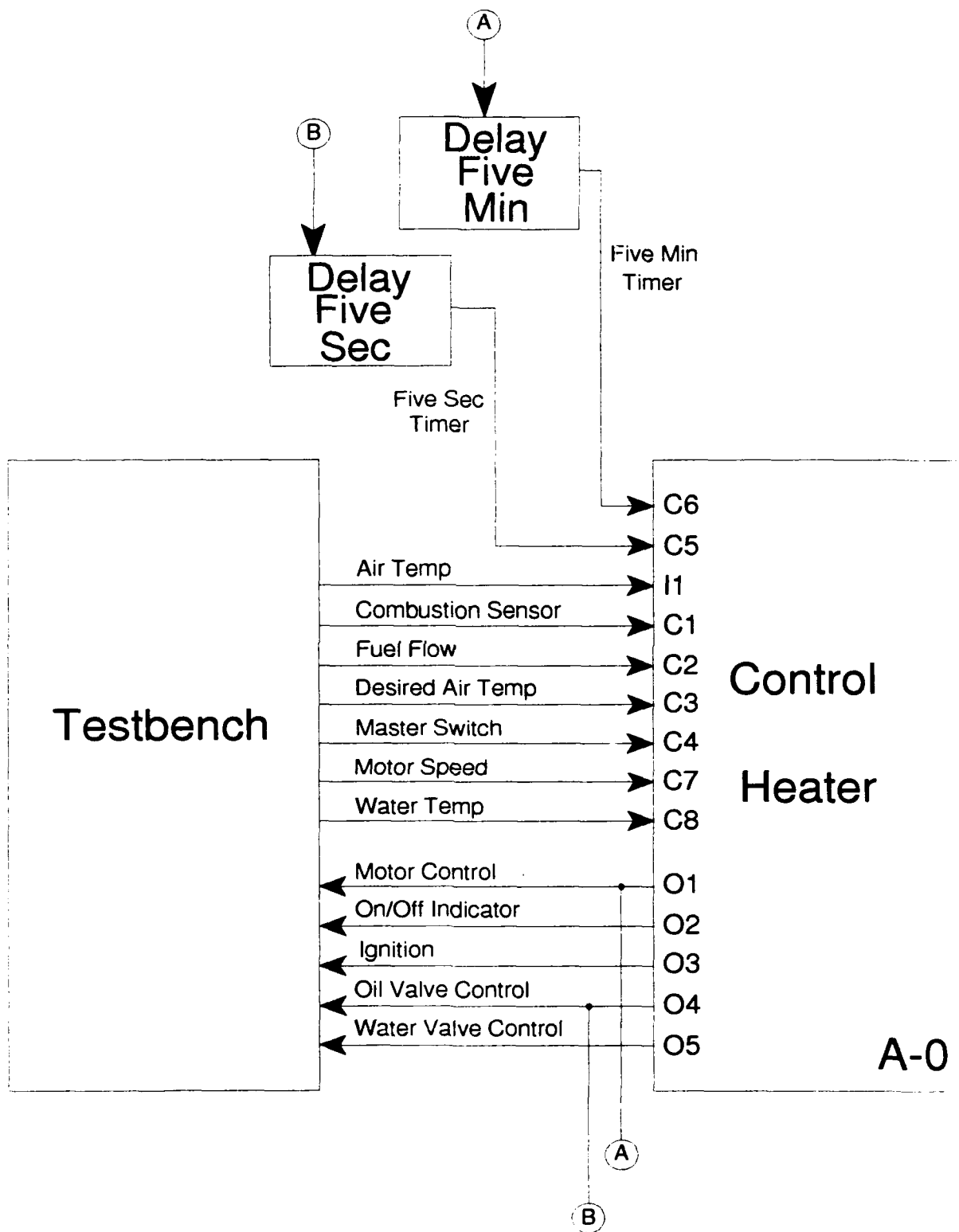


Figure 33. VHDL Test Environment connection to the SADT Model

VHDL simulation environment. The System entity declaration and architecture body for the Heating System problem is contained in Appendix C, Section C.2.

4.5.3 Component Configuration Declaration The component configuration declaration for the Heating System is located in Appendix C, Section C.15. This declaration specifies (by name) which library components to use for each component in the system under test.

4.6 Executing the VHDL Simulation of the Heating System

Sections 4.3, 4.4, and 4.5 described generation of the VHDL source code for the Heating System. The complete listing of the generated source is given in Appendix C. The next step in the process of validating the requirements specification contained in the SADT model is execution of the specification to observe the resulting behavior.

In VHDL, the execution of the specification is performed using the simulation environment. The source code is first compiled (*analyzed* in ZYCAD terminology) and then simulated. Once the code has successfully compiled, the VHDL simulator is invoked. The commands for compiling and simulating the VHDL model are dependent upon the environment used. In the ZYCAD environment used in this research, the command **zvan {source_file_name}** is used to invoke the compiler, and **zvsim {configuration_unit_name}** is used to invoke the simulator. Appendix C, Section C.16 provides the script file used to compile all the source code files of the Heating System. The configuration unit name for the Heating System was specified in the configuration declaration as *Struct_Config* (See Appendix C, Section C.15), therefore the simulator was invoked using **zvsim Struct_Config**. Discussion of specific commands used in the ZYCAD simulator are not described in this thesis. The reader is referred to the ZYCAD System VHDL User's Manual (55) and Reference Manual (54) for this information. Appendix C, Section C.19 contains an example output from a simulation session. An example section from this output is given in Figure 34. The output generated correlates directly to the tests contained in the testbench entity (see Appendix C, Section C.14). The text in the simulation output surrounded by a row of *********'s was manually generated using text output commands in the testbench.

The remaining text was generated by the simulator during execution. This output specifies

```
*****
TEST CASE 1: Initial Startup conditions tested. There
are three phases in the test. Requirements R1, R2, and R3
are tested.
*****

0 SEC
  DESIRED_AIR_TEMP      = 78
  AIR_TEMP              = 75
  MOTOR_CONTROL         = ON_STATE
  ON/OFF_INDICATOR      = ON_STATE
  FIVE_MIN_TIMER        = FALSE

*****
Passed TEST 1, PHASE 1.
*****
```

Figure 34. Sample VHDL Simulator Output

the values of the indicated signals at a specific simulation time. In the example given, Desired_Air_Temp has a value of 78, at 0 sec simulation time. These outputs were generated using the ZYCAD *monitor* routine in the simulation environment to display a signal and its value each time it changes value. In addition to the text based testing, the simulator can be run in a windowing environment. In the windowing environment, specific signals can be monitored on individual windows, and source code traces can be viewed on other windows. Several other debugging/monitoring options such as source code breakpoints, and line-by-line execution are available to the user, and are described in the ZYCAD User's Manual (55).

The robust simulation environment provided a valuable tool in validating the specification of the Heater System Problem. The behavior specified in the decision tables for this problem proved to be an accurate representation of the problem statement. The simulation did reveal, however, that the specification for the Compare Temperature activity shown in Table 9 was incomplete. The state where $T_A = T_R + 2$ was not covered in the behavior

Table 9. Incorrect Activity Behavior - A21

Compare Temperature				
Shut Down	Master Switch	Air Temp (T_A)	Des Air Temp (T_R)	Oil Enable
True				False
	Off			False
		$>T_R+2$		False
False	Heat		$>T_A+2$	True

definition, and needed to be added. The corrected behavior, $T_A \geq T_R + 2$, is shown in Table 10.

Table 10. Corrected Activity Behavior - A21

Compare Temperature				
Shut Down	Master Switch	Air Temp (T_A)	Des Air Temp (T_R)	Oil Enable
True				False
	Off			False
		$\geq T_R+2$		False
False	Heat		$>T_A+2$	True

4.7 The Lift Control System Problem

The Lift Control System problem (See Section 4.1) statement specifies the requirements for a Lift³ controller which schedules and controls one or more lifts in a multi-story building. The control aspect of the requirement includes the actual lift, the buttons and lights internal to the lift, and the buttons and lights on the floors of the building. The terms *Destination Buttons* and *Destination Lights* are used in the discussion to designate the buttons/lights internal to the lift. The terms *Summons Buttons* and *Summons Lights*

³(Lift is equivalent to Elevator. The term lift is used in this document because it was the terminology used in the original problem statement.

designate the buttons/lights on each floor. The reader is referred to Yourdon's (51) Appendix G for a case study of an Elevator problem similar to the Lift specification used in this research. The complete problem statement for the Lift Control System was given as follows:

1. Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the lift.
2. Each floor has two buttons (except ground and top floor), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The illumination is cancelled when a lift visits a floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled. The algorithm to decide which to service first should minimize the waiting time for both requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests (or model a 'holding' floor).
4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority (can this be proved or demonstrated?).
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel (can this be proved or demonstrated?).
6. Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status.

4.7.1 SADT Analysis Figures 36 through 41 illustrate the SADT representation for the lift system. The Lift problem was hierarchically decomposed into the tree structure shown in Figure 35. At the A-0 level, the Emergency Button, Summons Buttons, and Destination Buttons are the external constraints of the system. Light On, Light Off, Emergency Signal, Motor Indication and Lift Status are the outputs from the system. These arrows (data elements) are all *pinelined* arrows meaning that they represent more than one data item in a single arrow. Many of the arrows in the SADT specification for the Lift are this type of arrow. This allowed for a single arrow to represent information for more than one lift, and more than one floor.

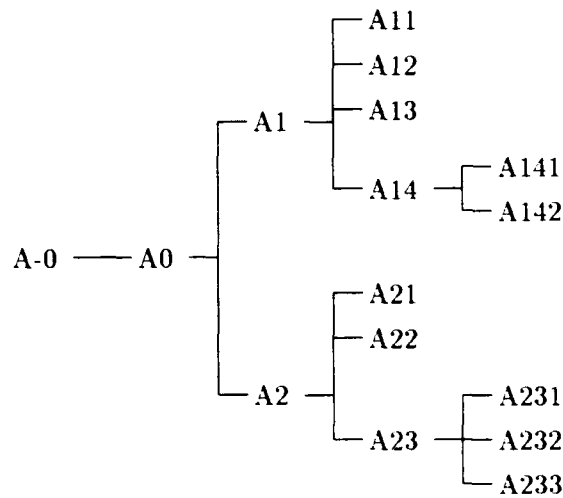


Figure 35. SADT Tree Structure for the Lift Problem

The A0 level breaks the system down into two activities, one to process the requests and one to schedule and implement events (Figure 37). Because of the dynamic behavior of the lifts, the Motor Indication (O4) and Lift Status (O5) information gets fed back to the Process Lift Requests activity (A1).

The A1 level represents a breakdown of the Process Lift Requests activity (A1) and is illustrated in Figure 38. Determine Status (A11) processes the Emergency Buttons (C2) and current Motor Indications (C1) to put the lifts in and out of service by setting the Current Status and Emergency Signal (O3) data elements. The Select Lift activity (A12) processes Summons Buttons (C3) and current Lift Status (I1) information and decides which lift will respond to a given summons. The Issue Sequence Commands activity (A13) takes the Summons Request data from the Select Lift activity (A12) and also any inputs from Destination Buttons (C4) and provides the Floor (O4), Lift (O5), and Direction (O6) data elements that are used by the Schedule Lift Events activity (A2). The A14 (Acknowledge Requests) activity processes turning the Lights On (O1) and Lights Off (O2).

Acknowledge Requests (A14) of Figure 38 is broken down in Figure 39 and has two

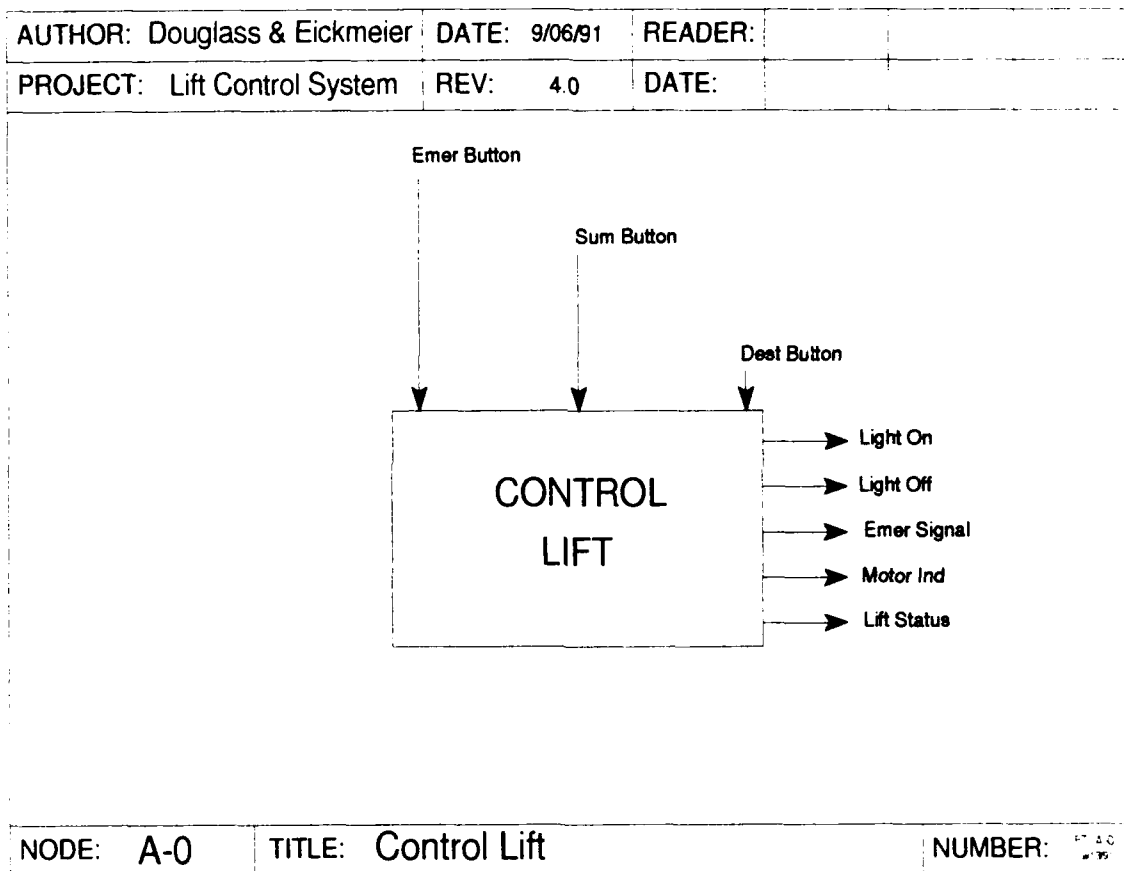


Figure 36. Lift Control System A-0 Diagram

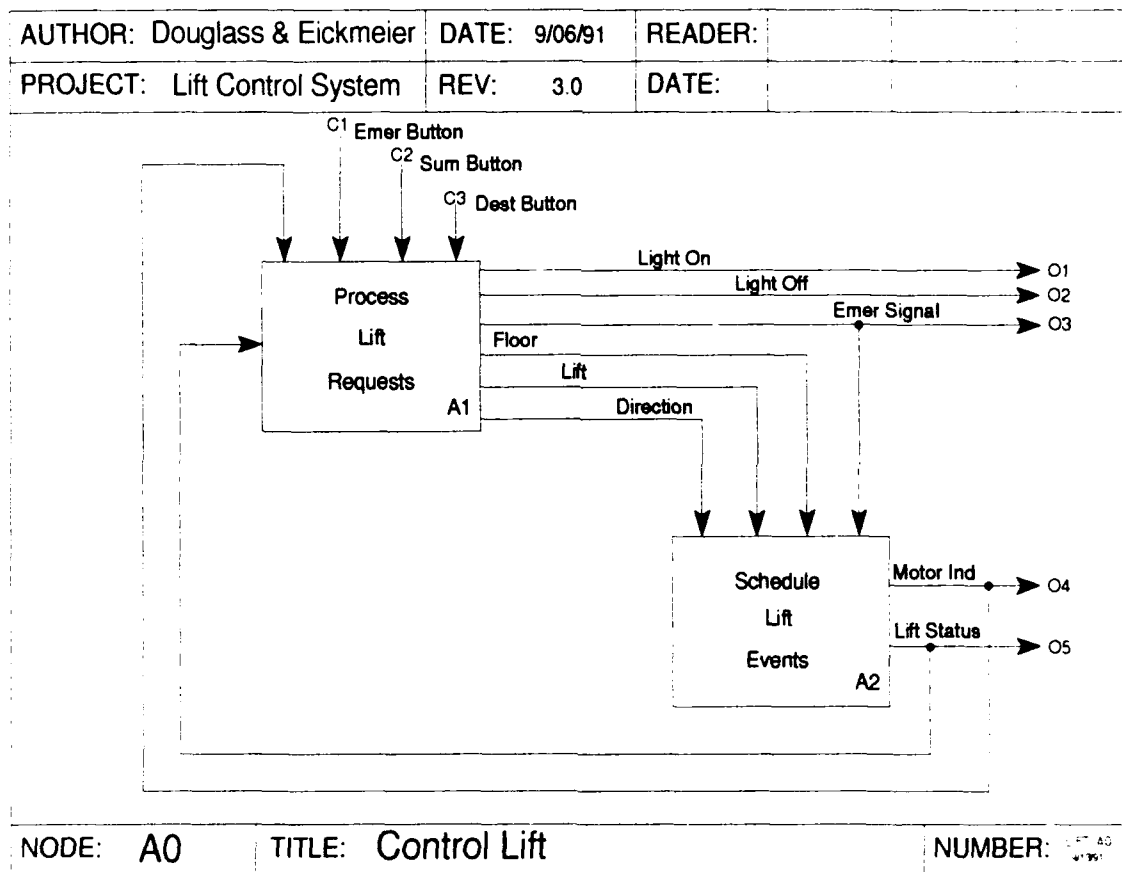


Figure 37. Lift Control System A0 Diagram

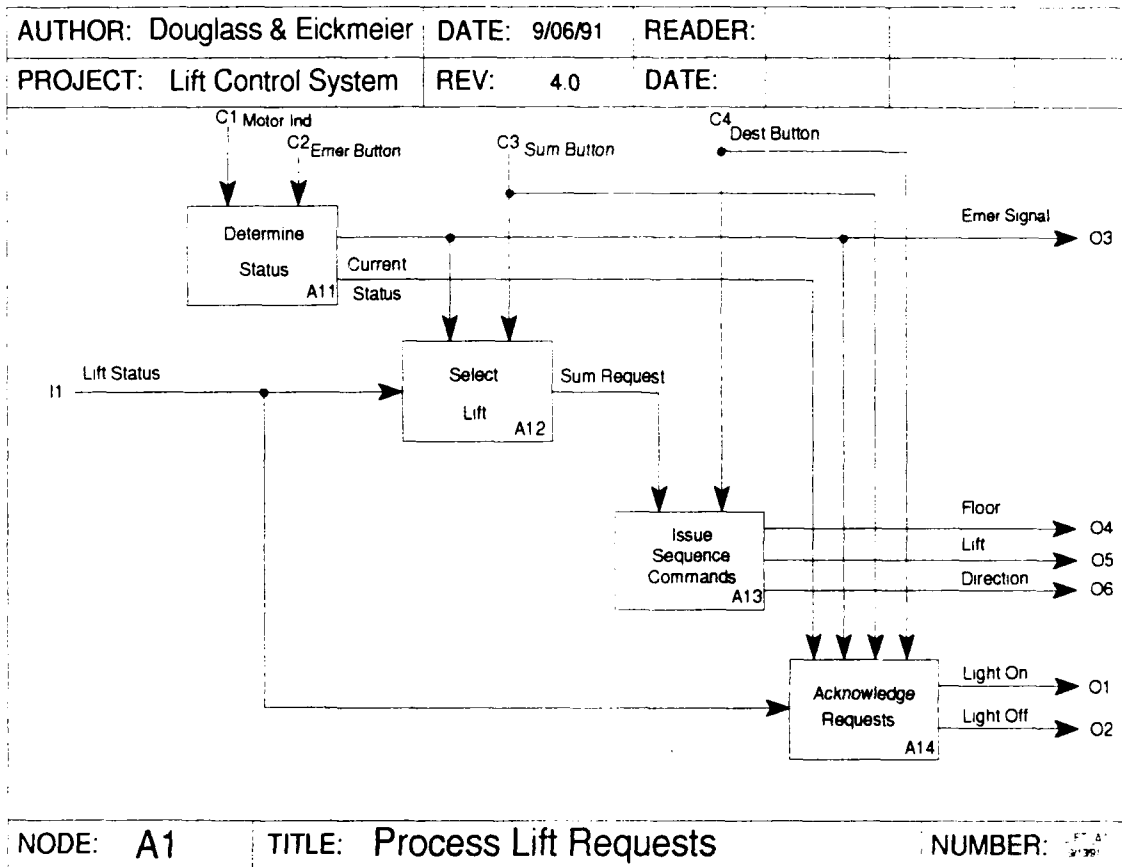
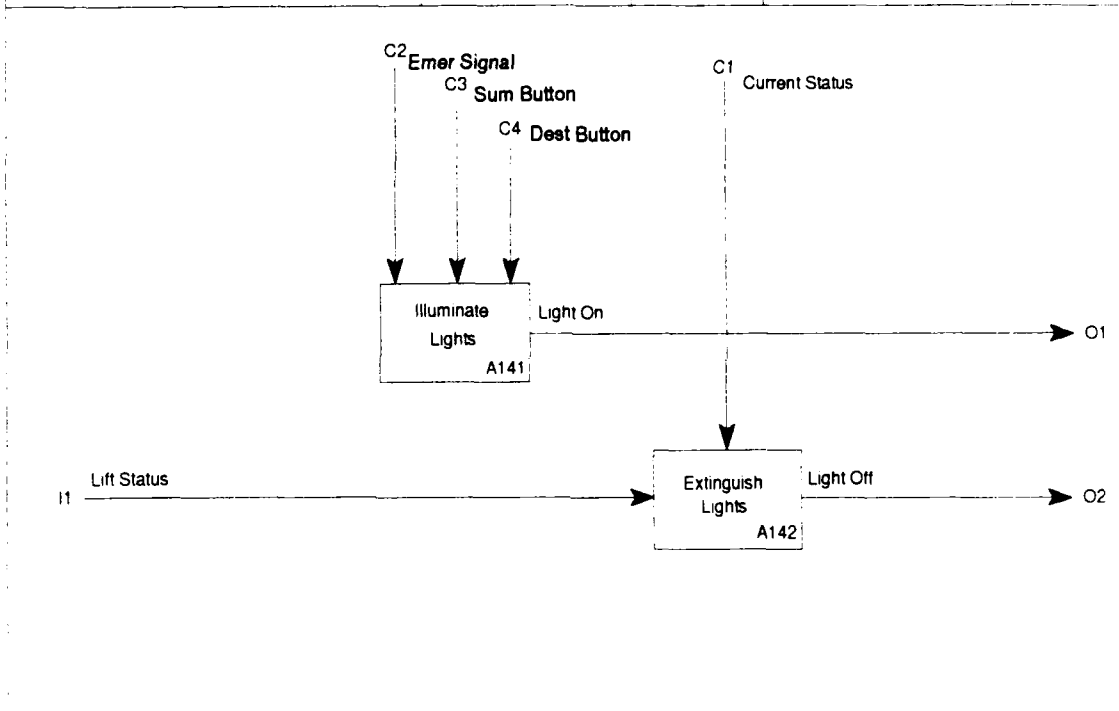


Figure 38. Lift Control System A1 Diagram

AUTHOR: Douglass & Eickmeier	DATE: 9/06/91	READER:			
PROJECT: Lift Control System	REV: 4.0	DATE:			



NODE: A14	TITLE: Acknowledge Requests	NUMBER: 17-114 9/91

Figure 39. Lift Control System A14 Diagram

activities, Illuminate Lights (A141) and Extinguish Lights (A142). The input buttons determine when lights are illuminated, and Current Status (C1) controls when they are extinguished.

The A2 activity (Schedule Lift Events) in Figure 37 breaks down into three activities (Figure 40). The Update Minimum/Maximum activity (A21) keeps track of the lowest/highest floor that is currently on a lift's schedule. Add Events (A22) adds new events to the lifts' current schedules. Process Lift Events (A23) processes the lifts' schedules by simulating movement and state changes of the lifts. The lifts' schedules are fed back from A23 to A21 via the New Sched 1 and New Sched 2 data elements. This is necessary because as the states and locations of the lifts change, their schedules must be updated accordingly. Process Lift Events (A23) also provide the external outputs of Motor Indication (O1) and Lift Status (O2).

Process Lift Events (A23) breaks down into three activities as illustrated in Figure 41. A separate activity is used for processing each lift. In this example, two lifts were addressed: thus activities A231 and A232 were created. The status of each lift is combined in the Set System Status activity (A233). Again feedback is used, to represent the dynamic behavior of the lifts.

At the A2 and A23 levels, the system has been scoped for a two lift and three floor system. However, the general breakdown of the system is easily expanded to any number of lifts and floors.

Appendix B, Tables 37 through 78 specify the behavior for the leaf node activities of the lift system. These decisions tables follow the standard orientation with data elements (variables) listed in the first column, and sets of conditions/actions listed in subsequent columns. This orientation was described in Section 3.5.2 and depicted in Figure 15 and Tables 3 and 4. The pre-conditions appear at the top of the table, and the post-conditions (actions) appear at the bottom. This orientation allows for longer lists of variables appearing on the same page. Multiple conditions are represented by breaking the table into several parts, with each part containing the variable names in column 1.

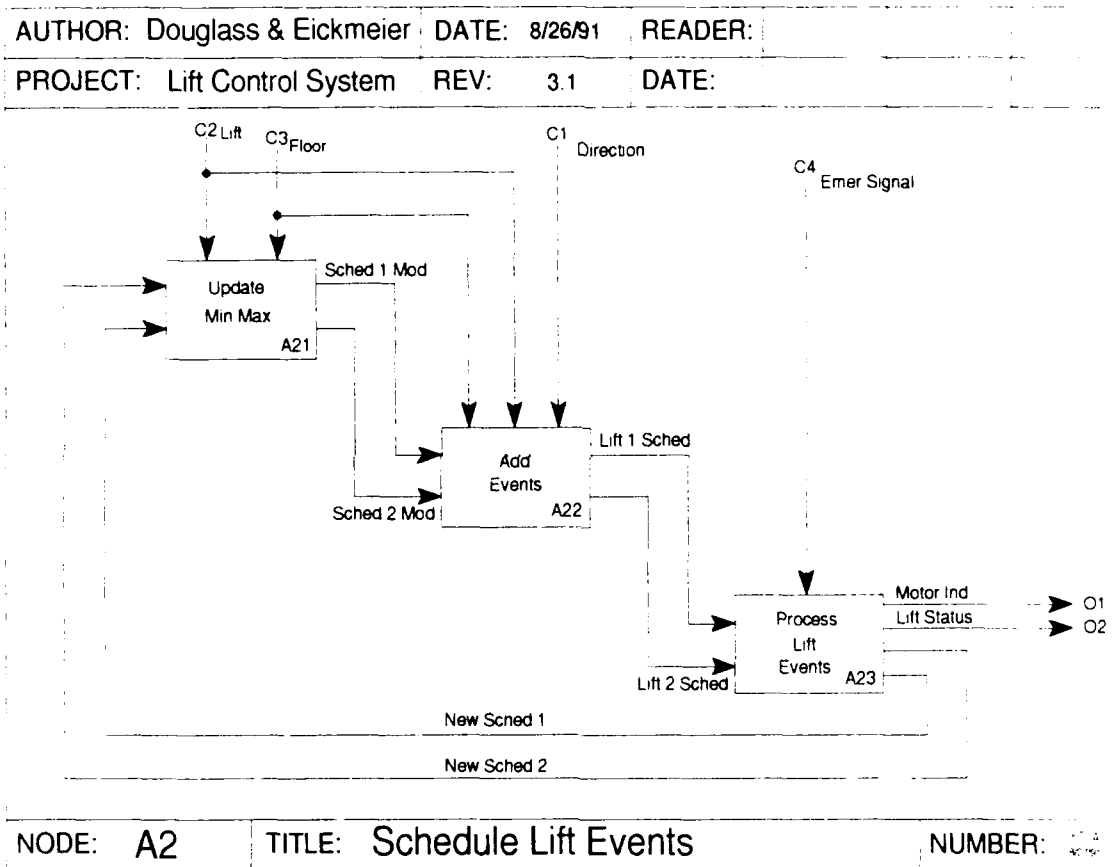


Figure 40. Lift Control System A2 Diagram

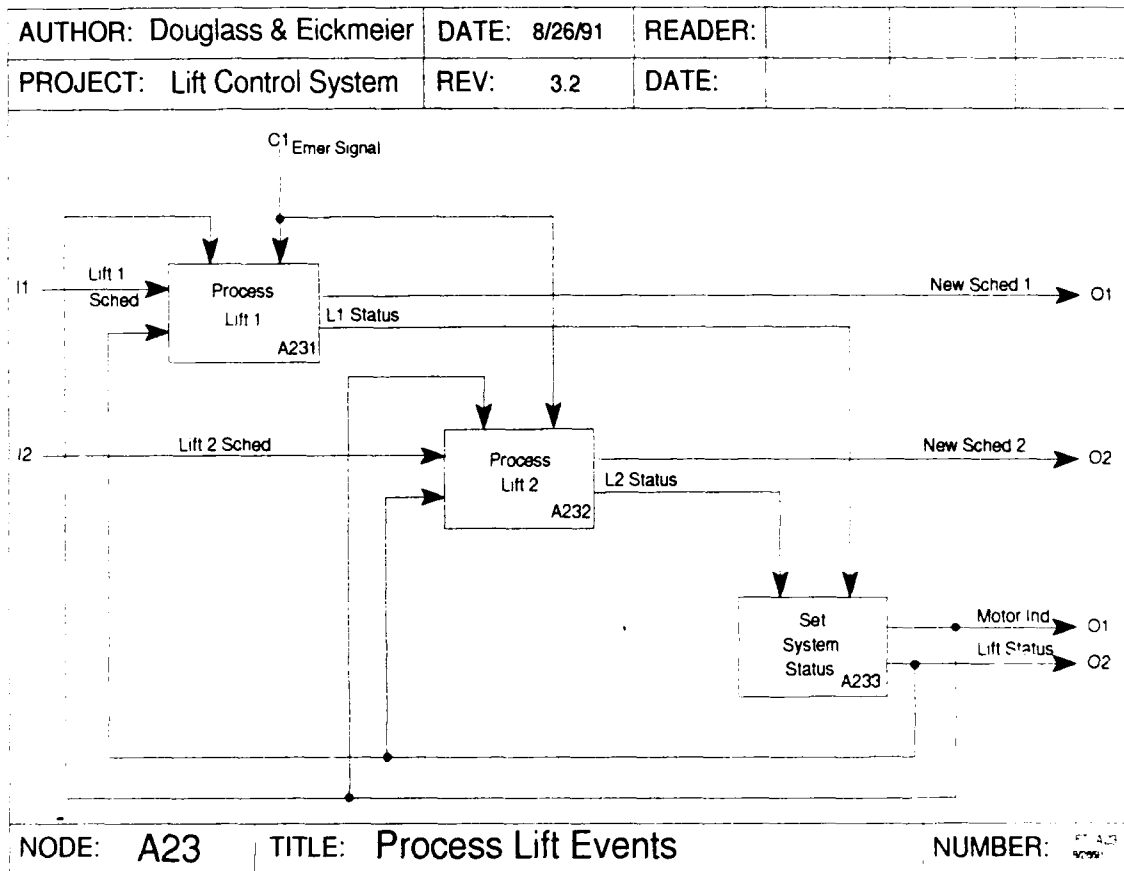


Figure 41. Lift Control System A23 Diagram

4.7.2 Problems Encountered The problem statement for the Lift Control System was more detailed in its definition than the Heating System problem statement. However, there were still requirements which were incomplete in the definition. The following items needed further clarification:

1. A lift is supposed to go into an 'out of service' state when an Emergency Button is pressed. Does this mean the lift stops immediately, or at the next floor in the direction of travel?
2. While a lift is in the 'out of service' state, are Destination Requests internal to the 'out of service' lift added to the schedule for that lift?
3. While a lift is in a 'holding' state, is the door on the lift open or closed? More specifically, does a Summons Request need to be issued first so the doors can open and a passenger enter the lift to input any Destination Requests?

These issues may be considered implementation details, but the following assumptions were made in order to generate an executable simulation of the Lift System behavior.

- A lift will stop at the next floor in the direction of travel when the Emergency Button for that lift is pressed.
- While in the 'out of service' state, Destination Requests which originate inside the 'out of service' lift will be added to the schedule for that lift.
- The lift doors will close while in the 'holding' state. A Summons Request will bring the lift out of the 'holding' state, and open the doors.

4.7.3 Test Cases Generated Six basic test cases were developed to test proper behavior of the Lift Control System in accordance with the problem statement. Again, the goal for developing these tests was not exhaustive testing of the Lift Control System. The tests exercise the functionality described in the problem statement. In each test case description, the expected results and the requirements tested are indicated using the paragraph numbering of the problem statement. The second paragraph in the problem statement contains two distinct requirements: 1) illuminating and extinguishing

lights, and 2) dealing with more than one request per floor, thus these two requirements are designated *2a* and *2b* in the test case descriptions. The expected results describe the expected behavior of the lifts only. To keep the discussion simple, the expected behavior of the lights is not described. This behavior is straight forward, and is easy to validate in the simulation output. Execution of these tests and the results achieved is described in Section 4.8.5.

1. TEST CASE 1. This test checks basic operation and ensures that the control system can handle both lifts moving at the same time. Requirements Tested: 1, 2a, 3, and 4. It consists of the following requested events:

- Up Summons from floor 2
- Down Summons from floor 3
- Destination Request for Lift 1 to floor 3
- Destination Request for Lift 2 to floor 1

Expected Results: Lift 1 should satisfy the Summons request on floor 2, and Lift 2 should satisfy the Summons request on floor 3. Once each lift has arrived at their summoned floor, the Destination Requests should be met. At the end of the test, Lift 1 should be in an Idle state at floor 3; Lift 2 should be Idle at floor 1. The term *Idle* or *Idle State* is used to describe the behavior described in problem statement paragraph 3.

2. TEST CASE 2. Test 2 puts Lift 2 out of service and tests Lift 1 for compliance with scheduling requirements. Requirements Tested: 1, 2a, 3, 5, and 6. The following requests make up test 2:

- Set Lift 2 Emergency Button to ON
- Destination Request for Lift 1 to floor 1
- Up Summons Request from floor 2
- Destination Request for Lift 2 to floor 1 (no action should occur, as Lift 2 is out of service)

Expected Results: Lift 2 should go into the 'out of service' state, and an Emergency Signal should be generated. Lift 1 responds to the destination request. While Lift 1 is moving from floor 3 to 1, an up summons request is given. The lift should not stop on the way down, but should go to floor 1 since the summons request is in the opposite direction of current travel. Lift 1 should satisfy the summons request on floor 2 after stopping on Floor 1. At the end of the test, Lift 1 should be Idle at floor 2 and Lift 2 out of service on floor 1.

3. TEST CASE 3. Test 2 checks the operation of the Lift Controller when both lifts are out of service. Requirement Tested: 6. The following requests make up test 3:

- Set both Lift Emergency Buttons to ON
- Down Summons Request from floor 3 (no action should occur)

Expected Results: Lift 1 should go into the out of service state. The summons request is ignored. At the end of the test, Lift 1 should be out of service at floor 2 and Lift 2 out of service on floor 1.

4. TEST CASE 4. Test 4 again checks for scheduling requirement compliance. This time Lift 1 is first sent down to floor 1. Then a Down Summons from floor 3 is requested followed by an Up Summons from floor 2. Requirements tested: 1, 2a, 3, 5, and 6. Test 4 is consists of the following events:

- Set Lift 1 Emergency Button to OFF
- Down Summons Request from floor 2
- Destination Request for Lift 1 to floor 1
- Down Summons Request from floor 3
- Up Summons Request from floor 2
- Destination Request for Lift 1 to floor 1

Expected Results: Lift 1 should go back in service and satisfy the summons request on floor 2. The Lift then travels to floor 1, switches direction and starts toward floor 3 to meet the Down Summons. The lift should stop at floor 2 on the way up to floor 3 to meet the Up Summons at floor 2. When Lift 1 reaches floor 3, it stops, changes directions, and heads for floor 1 to satisfy the last Destination Request. At the end of the test, Lift 1 is idle on floor 1, and lift 2 is out of service on floor 1.

5. TEST CASE 5. This test checks the system behavior when a request is issued for a floor where a lift is already at. Requirements Tested: 1, 2a, and 3. It consists of the following:

- Set both Emergency Buttons to OFF
- Up Summons Request from floor 1
- Destination Request for Lift 1 to floor 1

Expected Results: Both requests in this test should be automatically met, without moving either lift. At the end of the test, Lift 1 is idle on floor 1, and lift 2 is idle on floor 1.

6. TEST CASE 6. This test checks system behavior when both Summons Request Buttons for the same floor are pressed at the same time. In this test, Lift 2 is put out of service, thus only Lift 1 is responding to requests. Requirements Tested: 1, 2a, 2b, 3, 4, and 6. The test consists of the following:

- Set Lift 2 Emergency Button to ON

- Up Summons Request from floor 1
- Destination Request for Lift 1 to floor 3
- Down Summons Request from floor 2 before the Lift reaches floor 2.
- Up Summons Request from floor 2 after the Lift has passed floor 2, but before reaching floor 3.
- Destination Request for Lift 1 to floor 1 when Lift stops at floor 2 for the Down Summons Request.

Expected Results: Lift 1 should go back into service based upon the Up Summons Request, and start up toward floor 3 to satisfy the Destination Request. The Lift should not stop at floor 2 to satisfy the Down Summons Request because it is in the opposite direction of travel. The Lift should stop at floor 3, change direction, and go to floor 2 to satisfy the Down Summons. The Lift should then travel down to floor 1 to satisfy the Destination Request, change direction, and proceed to floor 2 to satisfy the Up Summons Request.

4.8 Code Generation and Simulation of the Lift Control System

The SADT specification for the Lift Control System introduced several constructs not used in the Heater System. These needed to be translated into VHDL. Specifically, these were Pipeline Arrows and Feedback Loops. These SADT constructs were defined in Section 3.2.3. In addition to the new SADT constructs, a problem was encountered with the concurrent processing of SADT activities. These differences are described further in the following sections.

4.8.1 Pipeline Arrows Pipeline arrows were used in the SADT description of the Lift Control System to represent collections of inputs, controls, and outputs using a single arrow. As an example, the Destination Button control arrow on the Lift Control System Context diagram (Shown again here in Figure 42) represents the value of the floor and the lift corresponding to the button pressed. Figure 43 shows how the Destination Button arrow can be visualized as two individual arrows, Floor and Lift, which merge to form the Destination Button arrow.

In this research, pipeline arrows were treated as individual arrows in the SADT diagrams. They were never split (or decomposed) into their individual components as allowed in the SADT methodology. The components of the pipeline arrows were referenced in the

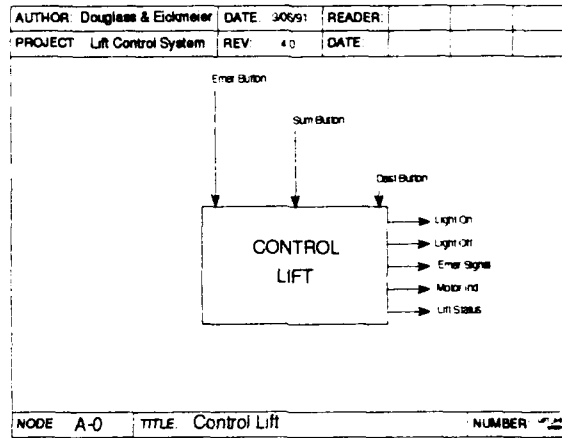


Figure 42. Lift Control System A-0 Diagram

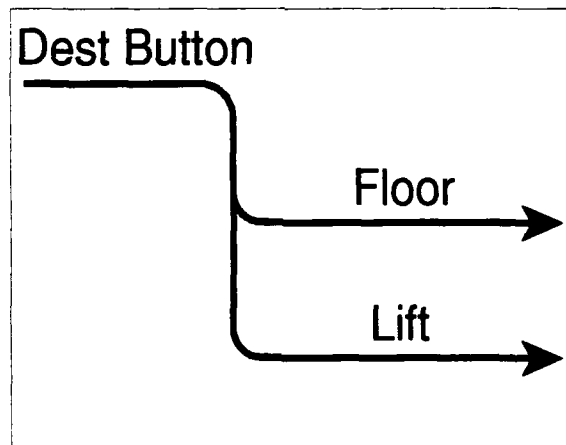


Figure 43. Pipeline Arrow Representation

decision tables using a 'dot' notation. To reference the Floor component of Destination Button, the notation `Dest_Button.Floor` was used. This notation is in the form: *Parent.Child*. Here the parent arrow is the main arrow in the pipeline, and the children are the subcomponents.

A VHDL record type was used to represent the pipeline concept. The record definition for the Destination Button arrow is given in Figure 44. This VHDL structure, and the

```

type Dest_Button_Type is record
  Floor : Floor_Type;
  Lift  : Lift_Type;
end record;

```

Figure 44. VHDL Record Type Definition of an SADT Pipeline Arrow

'dot' reference notation mapped directly to the SADT pipeline concept. Decomposition of pipeline arrows on the SADT diagrams can be supported using the same VHDL structure. However, the designer who creates the decision tables for an SADT diagram using *pipeline* decomposition must use the *Parent.Child* notation in the decision table entries, and for the arrow names on the diagram. An example of using this naming practice on a diagram is shown in Figure 45.

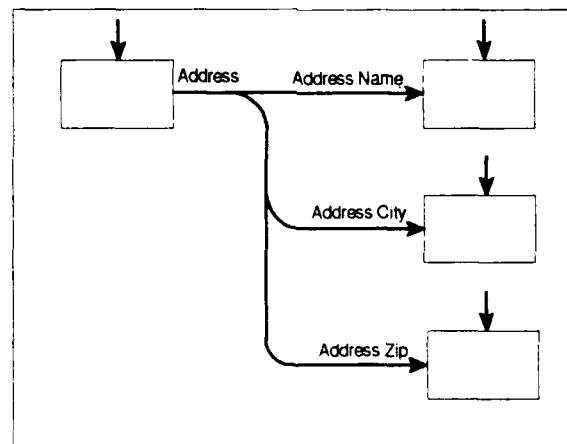


Figure 45. Pipeline Arrow Using Dot Notation

4.8.2 Feedback Loops The Lift Control System specification in SADT uses both Dataflow and Control Feedback (defined in Section 3.2.3) to model the dynamic nature of the system. Figure 37 shows two examples of the feedback used. The Lift Status arrow is a Dataflow feedback, and the Motor Indication arrow is a Control feedback. The only difficulty in implementing the VHDL code to capture the behavior of the feedback arrows arose when a feedback cycle existed in the SADT model. Figure 40 shows the activities involved in updating and adding events to the Lift schedules. The two schedules needed to be modified and passed to the next activity based upon the logic defined in the decision tables. In order to maintain the information in the schedules it was passed in a cycle from Activity A21 to Activity A22 to Activity A23 and then back to Activity A21. Since the arrows were each distinct objects in the SADT and VHDL representations, the information needed to be copied from object to object at each activity. For example the schedule, New Sched 1 enters Activity A21 as an input, is modified as appropriate, and exits as Sched 1 Mod. The logic to do this copy is shown in the partial decision table for Update Min/Max shown here in Table 11. The corresponding VHDL source code for the second column of this table is given in Figure 46.

```

1      elsif Floor >= New_Sched_1.Max and
2          Floor > New_Sched_1.Min and
3          Lift = 1 then
4          Sched_1_Mod.FL1_Stop <= New_Sched_1.FL1_Stop;
5          Sched_1_Mod.FL1_Dest <= New_Sched_1.FL1_Dest;
6          Sched_1_Mod.FL2_Stop <= New_Sched_1.FL2_Stop;
7          Sched_1_Mod.FL2_Dest <= New_Sched_1.FL2_Dest;
8          Sched_1_Mod.FL3_Stop <= New_Sched_1.FL3_Stop;
9          Sched_1_Mod.FL3_Dest <= New_Sched_1.FL3_Dest;
10         Sched_1_Mod.FL1_Up <= New_Sched_1.FL1_Up;
11         Sched_1_Mod.FL2_Up <= New_Sched_1.FL2_Up;
12         Sched_1_Mod.FL2_Down <= New_Sched_1.FL2_Down;
13         Sched_1_Mod.FL3_Down <= New_Sched_1.FL3_Down;
14         Sched_1_Mod.Max <= Floor;
15         Sched_1_Mod.Min <= New_Sched_1.Min;

```

Figure 46. VHDL Process Definition for Activity Update_Min/Max

Table 11. Activity Behavior - A21 Part 1

Update Min/Max			
New Sched 1.FL 1 Stop			
New Sched 1.FL 1 Dest			
New Sched 1.FL 2 Stop			
New Sched 1.FL 2 Dest			
New Sched 1.FL 3 Stop			
New Sched 1.FL 3 Dest			
New Sched 1.FL 1 Up			
New Sched 1.FL 2 Up			
New Sched 1.FL 2 Down			
New Sched 1.FL 3 Down			
New Sched 1.Max			
New Sched 1.Min			
Floor	0	$\geq$ New Sched 1.Max	$\leq$ New Sched 1.Min
Floor	0	$>$ New Sched 1.Min	$<$ New Sched 1.Max
Lift	0	1	1
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	New Sched 1.Max	Floor	New Sched 1.Max
Sched 1 Mod.Min	New Sched 1.Min	New Sched 1.Min	Floor

The necessity to copy each record components values of the schedule from the input to the output resulted in very large decision tables and corresponding VHDL code. Recommendations for reducing the size of these tables are included in Chapter VII. The complete set of decision tables for the Lift Control System is given in Appendix B, and the Source code is contained in Appendix D.

4.8.3 Concurrent Processing of SADT Activities Recall from Sections 3.3.1 and 4.3.3 that the VHDL construct chosen to model the behavior of an SADT activity was the *Process Block*. It was stated that each Process Block executes concurrently with all other Process Blocks. Using this information, the reader may have noticed a problem in the previous section's description of the feedback cycle. The feedback cycle shown in Figure 40 is actually a sequential process where Activity A21 needed to execute before Activity A22 which needed to execute before Activity A23. The problem which arose here was the propagation of two different values around the cycle. This occurred when two Activities, namely A21 and A22, were both responding simultaneously to changes in their Lift and Floor Control inputs. Both activities updated the schedule as needed, and copied the remaining OLD input schedule to the output schedule. Upon completion of their updates, both VHDL processes go back to an inactive state. However, now new values on the schedule have been introduced by both A21 and A22. Both of these schedules propagate around the feedback cycle, causing an infinite loop because the processes are fired each time any field in the input signal changes (an *event* occurs).

The best solution to this problem would have been to avoid sequential dependencies in the SADT design. Both Hartrum (21) and Marca and McGowan (32) warn against the use of implied sequence in SADT models. The solution taken in this research effort was the introduction of additional logic in the VHDL source code to eliminate the infinite feedback cycle. Figure 47 shows the modified process definition for the example previously given in Table 11 and Figure 46. Lines 1 through 36 of code contains the logic added to specify when the input schedule is copied to the output schedule. The logic allows only "new" (changed) values to be copied from the input (New Sched 1) to the output (Sched 1 Mod). This eliminates the infinite loop at the beginning of the cycle. Now when A21

```

1      if New_Sched_1.FL1_Stop'EVENT then
2          Sched_1_Mod.FL1_Stop <= New_Sched_1.FL1_Stop;
3      end if;
4      if New_Sched_1.FL1_Dest'EVENT then
5          Sched_1_Mod.FL1_Dest <= New_Sched_1.FL1_Dest;
6      end if;
7      if New_Sched_1.FL2_Stop'EVENT then
8          Sched_1_Mod.FL2_Stop <= New_Sched_1.FL2_Stop;
9      end if;
10     if New_Sched_1.FL2_Dest'EVENT then
11         Sched_1_Mod.FL2_Dest <= New_Sched_1.FL2_Dest;
12     end if;
13     if New_Sched_1.FL3_Stop'EVENT then
14         Sched_1_Mod.FL3_Stop <= New_Sched_1.FL3_Stop;
15     end if;
16     if New_Sched_1.FL3_Dest'EVENT then
17         Sched_1_Mod.FL3_Dest <= New_Sched_1.FL3_Dest;
18     end if;
19     if New_Sched_1.FL1_Up'EVENT then
20         Sched_1_Mod.FL1_Up <= New_Sched_1.FL1_Up;
21     end if;
22     if New_Sched_1.FL2_Up'EVENT then
23         Sched_1_Mod.FL2_Up <= New_Sched_1.FL2_Up;
24     end if;
25     if New_Sched_1.FL2_Down'EVENT then
26         Sched_1_Mod.FL2_Down <= New_Sched_1.FL2_Down;
27     end if;
28     if New_Sched_1.FL3_Down'EVENT then
29         Sched_1_Mod.FL3_Down <= New_Sched_1.FL3_Down;
30     end if;
31     if New_Sched_1.Max'EVENT then
32         Sched_1_Mod.Max <= New_Sched_1.Max;
33     end if;
34     if New_Sched_1.Min'EVENT then
35         Sched_1_Mod.Min <= New_Sched_1.Min;
36     end if;

37     elsif Floor >= New_Sched_1.Max and
38         Floor > New_Sched_1.Min and
39         Lift = 1 then
40         Sched_1_Mod.Max <= Floor;

```

Figure 47. Revised VHDL Process Definition for Update_Min/Max

and A22 are both simultaneously updating the schedule, they only introduce changes in the schedule fields they are responsible for modifying. These changes are then propagated around the feedback cycle. The individual activities copy the change only while it is "new" to the activity in the loop. Once each activity has copied the new values across to their output schedule, the cycle is broken.

This solution only works in this particular instance because the fields modified by A21 and A22 are different (mutually exclusive). A more general solution is needed to allow for sequential dependencies, where absolutely necessary, to be modeled in the VHDL simulation environment. Armstrong (1) uses a separate process block to determine the order of execution of the FETCH, EXECUTE, and INTERRUPT processes of a Mark 2 CPU simulation. This separate process block, called STATE, "implements the state movement of the processor" (1:107) between the three states FETCH, EXECUTE, and INTERRUPT. This concept of using a separate VHDL process may prove to be a general solution to model sequential behavior. The "state" process would conceptually contain the *when* behavior contained in an SADT Activation rule as described in Section 3.5.1. Additional research in this area is recommended.

4.8.4 Real-Time Extensions to the Lift Control System Section 4.4 discussed how the Heating System was extended to address timing requirements. The Lift Control System specification did not have any stated timing requirements, but reality warranted adding some. The Lift Control System was first specified in SADT and VHDL without any concern for timing. This allowed the logical behavior to be validated first. Once this was completed the VHDL portion of the specification was extended to address timing. While executing the VHDL simulation of the time independent behavior, the Lifts moved up and down between floors in zero simulation time. While this allowed for validation of the basic decision-table logic, it did not allow testing of events happening simultaneously. For example, when a summons request was made on a particular floor, the lift moved "instantly" to the floor responding to the request. In a more realistic scenario, it takes some amount of time before the lift arrives at the requested floor, and other requests can be issued during the traveling time of the lift. To accommodate validating this type of behavior, a delay was added to

the VHDL code which specified the moving of the lifts. Adding timing to the Lift Control System was approached differently than the Heating System. Recall that in the Heating System, the generation of the time dependent events was external to the Heating System environment. In the Lift Control System, the time dependent events were added internal to the system specification, specifically to the behavior of the Set System Status (A233) activity shown on Figure 41. The timing behavior was added to the Set System Status activity (A233) because it had the easiest behavior description of the activities in the schedule loop (A231, A232, A233). Both A231 and A232 have large decision tables, and timing behavior would have harder to add and debug in these activities. To introduce the real-time behavior, a ten second delay was added to the Motor Indication and Lift Status outputs whenever the decision table logic specified the location of the lift be incremented or decremented by one floor. The resulting VHDL source code for this activity is shown in Figure 48.

As stated in the discussion for the Heater System, adding timing to a VHDL model can not be easily addressed in the automated transformation of SADT to VHDL. This is primarily because of the VHDL syntax required to specify timing. While this syntax is not extremely difficult to understand, it is difficult to represent in a decision table format. It is assumed that this step will be done manually by a user who understands both the problem domain of the system being designed and the VHDL environment.

4.8.5 Executing the VHDL Simulation of the Lift Control System The Lift Control System specification was considerably larger and more complex than the Heating System. The specification involved more states in which the system could be in. A quick glance at the forty-two decision tables in Appendix B supports this claim. Because of the increase in size and complexity, there was a greater benefit realized in simulation of the specification. Section 4.7.3 outlined the tests performed on the Lift Control system. As described in Section 4.5.1, a VHDL Testbench was created to execute the tests on the Lift System. This Testbench is included for reference in Appendix F, Section E.1. An example output from execution of the simulation is contained in Appendix F, Section E.2. A significant contribution was identification of the infinite loop in the real time behavior of the system.

```

architecture Behavior of Set_System_Status is

    signal L1_Next_Floor : Boolean := False;
    signal L1_Prev_Floor : Loc_type := 1;

begin
    process (L1_Status, L2_Status) begin

        if L1_Status.Motor = Run and L1_Status.Motor'Event then
            Lift_Status.L1_Loc <= 0, L1_Status.Loc after Run_Delay;
            Lift_Status.L1_Dir <= L1_Status.Dir;
            Motor_Ind.L1_Ind <= L1_Status.Motor;
            L1_Next_Floor <= False, True after Run_Delay;
            L1_Prev_Floor <= L1_Status.Loc;

        elsif L1_Status.Motor = Run and not L1_Status.Motor'Event and
            L1_Status.Loc'Event and L1_Next_Floor = True then
            Lift_Status.L1_Loc <= 0, L1_Status.Loc after Run_Delay;
            Lift_Status.L1_Dir <= L1_Status.Dir;
            Motor_Ind.L1_Ind <= L1_Status.Motor;
            L1_Next_Floor <= False, True after Run_Delay;
            L1_Prev_Floor <= L1_Status.Loc;

        elsif L1_Status.Motor = Run and not L1_Status.Motor'Event and
            L1_Next_Floor = False then
            Lift_Status.L1_Dir <= L1_Status.Dir;
            Motor_Ind.L1_Ind <= L1_Status.Motor;

        elsif L1_Status.Motor = Stop and L1_Status.Motor'Event and
            L1_Next_Floor = False then -- Emergency Condition
            Lift_Status.L1_Loc <= L1_Prev_Floor;
            Motor_Ind.L1_Ind <= L1_Status.Motor;
            Lift_Status.L1_Dir <= L1_Status.Dir;
            L1_Next_Floor <= False;

        else -- Normal Stop or Idle
            Motor_Ind.L1_Ind <= L1_Status.Motor;
            Lift_Status.L1_Loc <= L1_Status.Loc;
            Lift_Status.L1_Dir <= L1_Status.Dir;
        end if;
    end process;
end Behavior;

```

Figure 48. VHDL Code for Adding Timing to the Lift

This problem was discussed in Section 4.8. The next significant finding was a failure to specify the desired behavior for all the system states. Simulation revealed valid states of the Lift System which were not included in the decision tables for the system. These states were added to the decision tables to correct the oversight. Both of these problems were designer error. A similar finding was the need to specify an initial state of the Lift Control System. Without an initial state specified, the Lift System reacted to whatever value was present at the beginning of the simulation. This behavior was often undesirable. Initial conditions were not specified in the original problem statement, and reflect an incompleteness in the specification. Initial conditions were added to the VHDL code to correct this oversight as shown in Figure 49. The syntax for assignment of an initial value is the same as used in the Ada language.

```
entity Set_System_Status is
    generic (Run_Delay : Time := 10 sec);
    port ( L1_Status    : in L1_Status_Type;
          L2_Status    : in L2_Status_Type;
          Motor_Ind    : out Motor_Ind_Type := (Idle,Idle);
          Lift_Status  : out Lift_Status_Type := (Up,1,Up,1));
end Set_System_Status;
```

Figure 49. VHDL Code for Specifying Initial Conditions in a Record Structure

4.9 Steps Taken in Transforming SADT into VHDL Source Code

This section summarizes the steps taken to transform an SADT specification into an executable VHDL specification. Most of these steps could be performed by an automated transformation system. Chapter V outlines a design for such an automated transformation. Specifically, Section 5.5 discusses each of the following steps, and describes the feasibility of automating each step. Automation of Steps 6 and 9 of the process are not addressed in the design, however, potential methods for automating these steps are proposed as additional areas of research in Section 7.4.

1. Generate a package containing type definitions for the SADT model.
2. Generate a VHDL entity declaration for each SADT activity box.
 - (a) Use the Inputs and Controls as the **in** ports in the entity port declaration.
 - (b) Use the Outputs as the **out** ports in the entity port declaration.
3. Generate a behavioral VHDL *Architecture Body* for each leaf-node activity.
 - (a) Generate the *Declaration Statement* for the body.
 - (b) Generate a *Process Block* within the Architecture Behavior. Sensitize the *Process Block* to the INPUTs, and CONTROLs of the leaf-node activity.
 - (c) Read the behavior definition for the leaf node activity from the decision tables and create the corresponding if-then-elsif construct in the *Process Block*.
4. Generate a structural VHDL *Architecture Body* for each parent SADT activity box.
 - (a) Generate the *Declaration Statement* for the body.
 - (b) Generate Component Declarations in the Architecture Structure for each child of the activity being defined.
 - (c) Generate a VHDL *Signal* declaration for each non-boundary arrow present in the child activity.
 - (d) Generate component instantiations for each child activity.
 - (e) Generate port maps for the instantiated components using the rules stated in Step 2, and the local signals generated in Step 4c. This step is logically equivalent to wiring the VHDL components together with signals.
 - (f) Determine if any Output boundary arrows were replaced in the component port maps (generated in Step 4e) with one of the local signals generated in Step 4c. If so, generate a signal assignment statement for each occurrence to assign the value of the local signal to the boundary arrow.
5. Generate a VHDL entity declaration for testbench entity.
 - (a) Use the Inputs and Controls of the corresponding A-0 entity (the UUT) as **out** ports in the testbench entity port declaration.
 - (b) Use the Outputs of the corresponding A-0 entity (UUT) as the **in** ports in the testbench entity port declaration.
6. Generate a behavioral VHDL *Architecture Body* for the testbench entity containing the desired test behavior.
7. Generate a VHDL entity declaration for the system entity. This declaration will not have an associated port map.
8. Generate a VHDL structural *Architecture Body* declaration for the system entity.
 - (a) Generate the *Declaration Statement* for the body.

- (b) Generate component declarations for the VHDL entity which corresponds to the A-0 SADT activity, and for the testbench entity.
- (c) Generate local VHDL *Signals* for each of the testbench ports.
- (d) Generate component instantiations for each component. Standard names like *UUT* and *Testbench* can be used for the SADT system and testbench components.
- (e) Generate port maps for the instantiated components using the local signals created in Step 8c.

9. Generate the system *Component Configuration* declaration.

4.10 Summary

This chapter has described the process taken to accomplish the following two objectives of this research effort: 1) validate the SADT to VHDL mapping as defined in Chapter III, and 2) evaluate the benefits of simulating a specification using VHDL.

The first goal was accomplished by applying the mapping definition to the Heating System and Lift Control System problems. While these are only two small pedagogical problems, the mapping definition proved to be valid for these cases. A step by step transformation procedure was formed from the SADT to VHDL mapping process which will serve as the starting point for defining an automated transformation algorithm described in Section 5.5. The VHDL hierarchical model and its constructs for specifying the low level behavior of a system map nearly one-to-one to the extended SADT model of this research. The VHDL model resulting from the mapping allows the designer visibility of the behavior of the SADT model at any level in the hierarchy.

Extension of the Heater and Lift problems to address time-related and concurrent behavior was also shown to be feasible for these two problems. The ability to test these features of a system allows a larger class of problems which can be validated using the VHDL executable simulation environment.

The second goal of evaluating the benefits of simulating a specification was realized during the execution of the two example problems. In the Heater and Lift simulations, both designer and specification errors in the specified behavior were revealed. Eliminating

these errors at specification time prevents the costly changes required when errors are not found until testing of the implemented system.

In addition to the errors revealed during simulation, the initial process of specifying systems with sufficient detail to make the behavior executable forces a more in-depth analysis of the requirements of the system. The specification of the Heating System was a prime example. Section 4.2.2 outlined the inconsistencies, and incompleteness of the problem statement. As a result of the analysis performed to develop the SADT specification, these inconsistent and incomplete areas were identified and corrected. This resulted in a better understanding of the problem being analyzed, and an unambiguous specification of the desired behavior of the system.

V. Requirements Analysis and Design of an SADT to VHDL Transformer

5.1 Introduction

The preceding chapters discussed the SADT to VHDL translation process and a manual implementation of that process with two example problems. This chapter will discuss what is required to implement an automated translation program. This program will be called the SADT Transformer. This transformation is designed in a general sense so both the Refine transformation of Douglass (16) and the VHDL transformation of this thesis can use it.

5.2 The Existing Model

The SADT Essential Model as implemented by Tevis (48) and Kitchen (27) is considered the baseline from which to begin. Douglass (16) has a detailed description of the entities and operations of this Essential Model in Appendix A of his thesis. An extension to the Essential Model is being worked on in a concurrent thesis effort by Blankenship (9). This extension is called the Abstract Model Manipulator since the model has been generalized to represent Data Flow Models, Concept Maps, State Transition Models, and Entity-Relationship Models in addition to the SADT method. The design contained in this chapter assumes the basic operations of Tevis and Kitchen's Essential Model have not changed. The terminology Abstract Model Manipulator (AMM), Abstract Model (AM) and Essential Model will be used synonymously.

The Ada programming language is assumed as the implementation language of this SADT Transformer, since that is the language of the existing Essential Model. The objects and operations of the Essential Model are used in this Transformer wherever possible and are referenced by name. Douglass's Appendix A (16) is the most concise reference for these names.

5.2.1 Addition Needed to the Abstract Model One addition to the Abstract Model's *Data-Element-Class* object is required by this Transformation process. The existing Abstract Model's Data-Element object is shown in Figure 50 and provides components for

defining the type information of the Data-Element. However, it does not provide a means to assign the Data-Element an *Initial Value*. This *Initial Value* is needed by both the Refine Transformation of Douglass (16) and the VHDL Transformation in this research so it should be added to this object. The addition to the Data-Element Record proposed is shown in bold type in Figure 51.

After this addition, the existing Abstract Model code will be used as part of the SADT Transformer by WITHing the appropriate operations.

5.3 Requirements for the SADT Transformer

The SADT Transformer must perform the following steps:

1. Read in a project (load)
2. Process the data elements by creating the necessary variable declarations
3. Process the information associated with the activities
4. Process the decision tables associated with the Leaf Node activities
5. Generate VHDL source code from the Abstract Model information.

Step 1 through Step 4 are common operations for both the Refine and VHDL translations. These operations also have to interface with the Abstract Model Manipulator. Therefore, these operations will be encapsulated into a single package called the AM-Reader. The Refine Translation operation and the VHDL Translation operation can interface directly with the AM-Reader, and do not need to interface with the Abstract Model Manipulator. Figure 52 illustrates the relationships between the objects. The following sections discuss the SADT Transformer process in greater detail. First, a design of the AM-Reader object is presented. This section proposes local data objects and operations for handling the data elements, activities, and decision tables of the Abstract Model. Finally, the actual VHDL source code generation is discussed.

Name:	Data-Element-Name-Type:= Null-Data-Element-Name;
Data-Type:	Data-Element-Data-Type:= Null-Data-Element-Data-Type;
Minimum:	Data-Element-Value-Type:= Null-Data-Element-Value;
Maximum:	Data-Element-Value-Type:= Null-Data-Element-Value;
Data-Range:	Data-Element-Value-Type:= Null-Data-Element-Value;
Values:	Environment-Types.Data-Buffer-Package.Manager-Type;
Description:	Environment-Types.Text-Buffer-Package.Manager-Type;
Reference:	Environment-Types.Text-Buffer-Package.Manager-Type;
Reference-Type:	Environment-Types.Reference-Type:= Environment-Types.Null-Reference-Type;
Version:	Data-Element-Version-Type:= Null-Data-Element-Version-Number;
Version-Changes:	Environment-Types.Text-Buffer-Package.Manager-Type;
Date:	Environment-Types.Date-Type:= Environment-Types.Null-Date;
Author:	Environment-Types.Author-Name-Type:= Environment-Types.Null-Author-Name;

Figure 50. Existing Data Dictionary Entry Format for a Data Element (27:55)

Name:	Data-Element-Name-Type:= Null-Data-Element-Name;
Data-Type:	Data-Element-Data-Type:= Null-Data-Element-Data-Type;
Minimum:	Data-Element-Value-Type:= Null-Data-Element-Value;
Maximum:	Data-Element-Value-Type:= Null-Data-Element-Value;
Data-Range:	Data-Element-Value-Type:= Null-Data-Element-Value;
Initial-Value:	Data-Element-Value-Type:= Null-Data-Element-Value;
Values:	Environment-Types.Data-Buffer-Package.Manager-Type;
Description:	Environment-Types.Text-Buffer-Package.Manager-Type;
Reference:	Environment-Types.Text-Buffer-Package.Manager-Type;
Reference-Type:	Environment-Types.Reference-Type:= Environment-Types.Null-Reference-Type;
Version:	Data-Element-Version-Type:= Null-Data-Element-Version-Number;
Version-Changes:	Environment-Types.Text-Buffer-Package.Manager-Type;
Date:	Environment-Types.Date-Type:= Environment-Types.Null-Date;
Author:	Environment-Types.Author-Name-Type:= Environment-Types.Null-Author-Name;

Figure 51. Proposed Data Dictionary Entry Format for a Data Element (27:55)

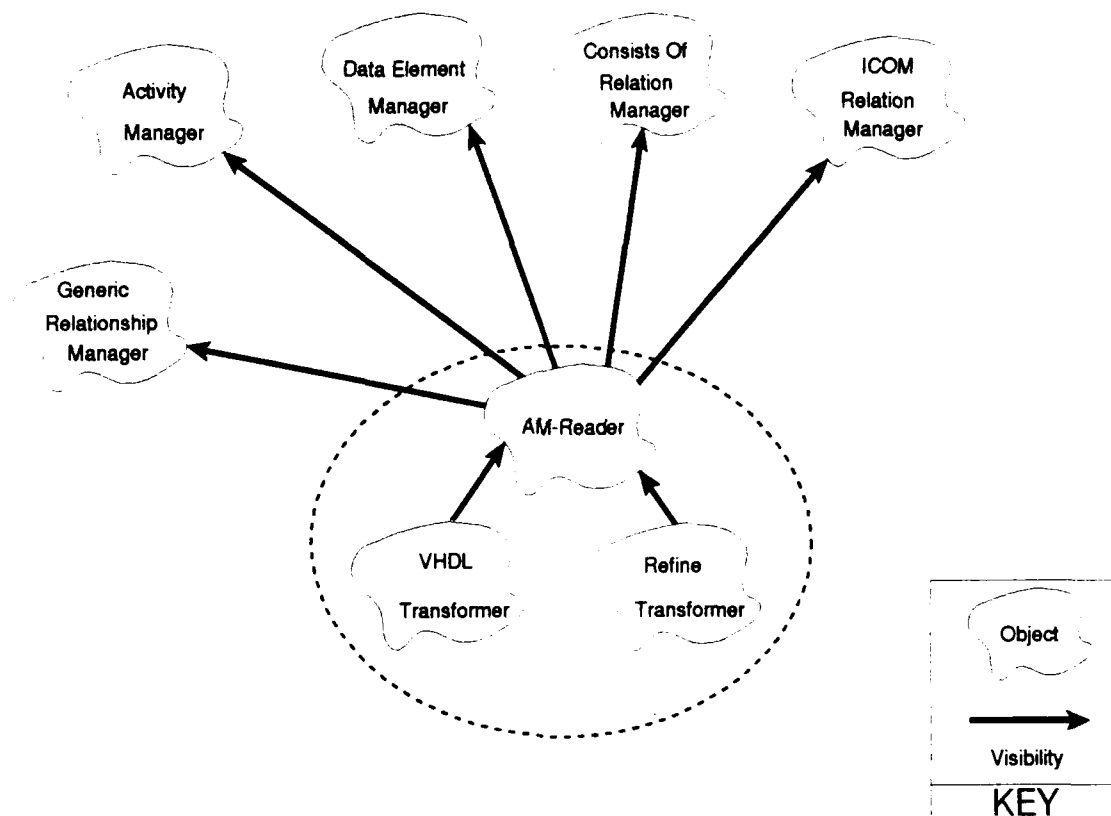


Figure 52. SADT Transformer Interfaces with the Abstract Model

5.4 Design of the AM-Reader Object

This section presents the design of the AM-Reader which performs Steps 1-4 of the SADT Transformation Process.

5.4.1 Reading in an Existing Project The first thing the AM-Reader must do is load a current SADT model into the system from the Abstract Model. A call to the existing procedure *ES-LOAD-SAVE.Project-Load-All* accomplishes the loading. The name of the project is passed in as a parameter.

5.4.2 Processing the Data Elements The data elements must be processed in order to create the necessary variable declarations. Within the AM-Reader, a local data structure is created to hold the necessary information about the data elements. This data structure can be implemented as a generalized linked list (23), and will contain the following attributes:

- NAME – implemented as a string.
- TYPE – implemented as a string.
- INIT – captures the initialization value, implemented as a string.
- CHILDREN – if the TYPE is record, then this field is a Linked List containing the NAMES of all of the composite parts. The individual NAMES are implemented as strings. Otherwise, this field is blank.
- VALUES – contains all the possible values for enumerated types. Implemented as a linked list of strings.

This data structure needs operations similar to the ones provided by each of the entities of the existing Abstract Model:

- Clear-Data-Element
- Create-Data-Element
- Set-Data-Element-NAME

- Set-Data-Element-TYPE
- Set-Data-Element-INIT
- Set-Data-Element-CHILDREN
- Value-of-Data-Element-At-Iterator
- Reset-Data-Element-Iterator
- Advance-Data-Element-Iterator
- Data-Element-Iterator-Done

5.4.2.1 Filling in the Data Element Data Structure The NAME, TYPE, INIT, and VALUES attributes will be filled in by one iteration through the data elements entity of the existing Abstract Model. First, the iterator is reset with the *Reset-Data-Element-Iterator* function. Next, the *Value-of-Data-Element-At-Iterator* function is used to retrieve an entire data element record that is currently pointed to by the iterator. Now the NAME, TYPE, INIT, and VALUES fields can be copied to the local data structure. Next, the *Advance-Iterator-To-Next-Data-Element* function is used to move to the next data element. This process continues until the iterator has reached the end of the data elements.

5.4.2.2 Completing the Data Element Data Structure The CHILDREN list still needs filling in. Since it is only required for data elements that are of TYPE Record, the local structure will be iterated through, looking for the TYPE field = Record. For each record that is found, the Abstract Model's *Consists-Of* entity will be iterated looking for the Parent-Child tuples who's Parent attribute matches the Parent attribute of the local structure who's TYPE = Record. Each tuple's Child attribute indicates that that NAME is to be added to the CHILDREN list of the local structure. The *Consists-of-Relationship-Class* of the existing Abstract Model has the necessary operations to iterate through, and obtain this information. The *Reset-Consists-of-Relation-Tuple-Iterator* resets the iterator, and the *Advance-Iterator-to-Next-Consists-of-Relation-Tuple* moves the iterator to the next tuple. The *Value-Of-Consists-Of-Relation-Tuple* operation returns the entire tuple pointed to by the iterator.

5.4.3 Processing the Activities This section contains the requirements to process the SADT Activities. These requirements include what is necessary for both the Refine and VHDL transformations. Although many of the operations are not required for the Refine transformation, the decision was to design an AM-Reader that meets the requirements of both research efforts. Therefore, the following discussion covers the complete Activity Reader Object, and is not limited to the Refine or VHDL requirements.

An SADT model is a hierarchical decomposition of a system from a single activity node into lower levels of activities which contain greater detail. This model can easily be represented as a tree structure as shown in Figure 53. Building a representation such as

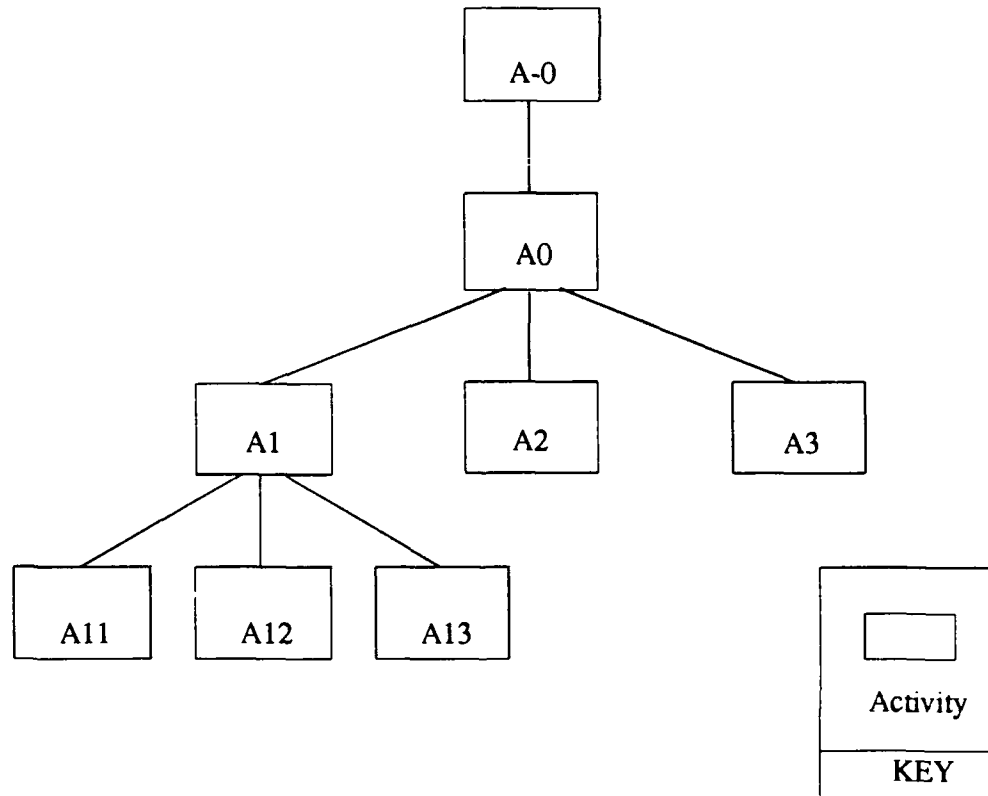


Figure 53. Example SADT Hierarchy

this in the AM-Reader simplifies the generation of the VHDL source code. The VHDL transformation requires the hierarchical structure be generated so a hierarchical VHDL

model can be built. The tree structure also identifies the leaf node activities needed by the Refine transformation.

The information necessary to build the SADT tree structure is contained in the Abstract Model. To obtain the information, however, several operations and iterations are required. Therefore, a set of operations in the AM-Reader are needed to Interface with the Abstract Model Manipulator. These operations will be called the Activity Reader.

5.4.3.1 The Activity Reader Data Object The Refine transformation process requires information about each leaf node activity in the SADT model. Once the leaf node has been identified, Refine needs the following information about the activity:

- Activity Name

The VHDL transformation process requires additional information about each SADT activity to support the hierarchical implementation chosen in this research. The following information is needed:

- Activity Name
- Activity Number
- List of Inputs
- List of Outputs
- List of Controls
- List of Children (if any)
- Parent Activity's Inputs, Outputs, and Controls

The required Activity information and the hierarchical structure of the SADT model can be represented as a generalized linked list of records. (23) The following record fields would provide the necessary information:

- NAME - implemented as a string.
- NUMBER - implemented as a string.

- **INPUTS** - implemented as a linked list of strings containing all the Inputs of the Activity.
- **CONTROLS** - implemented as a linked list of strings containing all the Controls of the Activity.
- **OUTPUTS** - implemented as a linked list of strings containing all the Outputs of the Activity.
- **CHILD** - If the Activity is decomposed further, this field would point to the first Child Activity. Otherwise, the field should be null signifying this Activity is a leaf node activity.
- **SIBLING** - A pointer to the next Activity at the same level of decomposition.
- **PARENT** - If the activity is a Child activity. This field points to it's Parent Activity, otherwise, it is null.

Figure 54 depicts the structure of the SADT model represented as a generalized list. Only the Child and Sibling Links are shown for simplicity of the diagram.

5.4.3.2 Operations Required by the Activity Reader Building the tree structure, and extracting the appropriate information to populate the Activity data structure will require multiple iterations through the Abstract Model's *Activity-Class* and *ICOM-Relation-Class*. The operations required of the *Activity-Class* are:

- Reset-Activity-Iterator
- Value-of-Activity-At-Iterator
- Advance-Iterator-To-Next-Activity
- Activity-Is-Child

and the *ICOM-Relation-Class* are:

- Reset-ICOM-Relation-Tuple-Iterator
- Value-of-ICOM-Relation-Tuple-At-Iterator

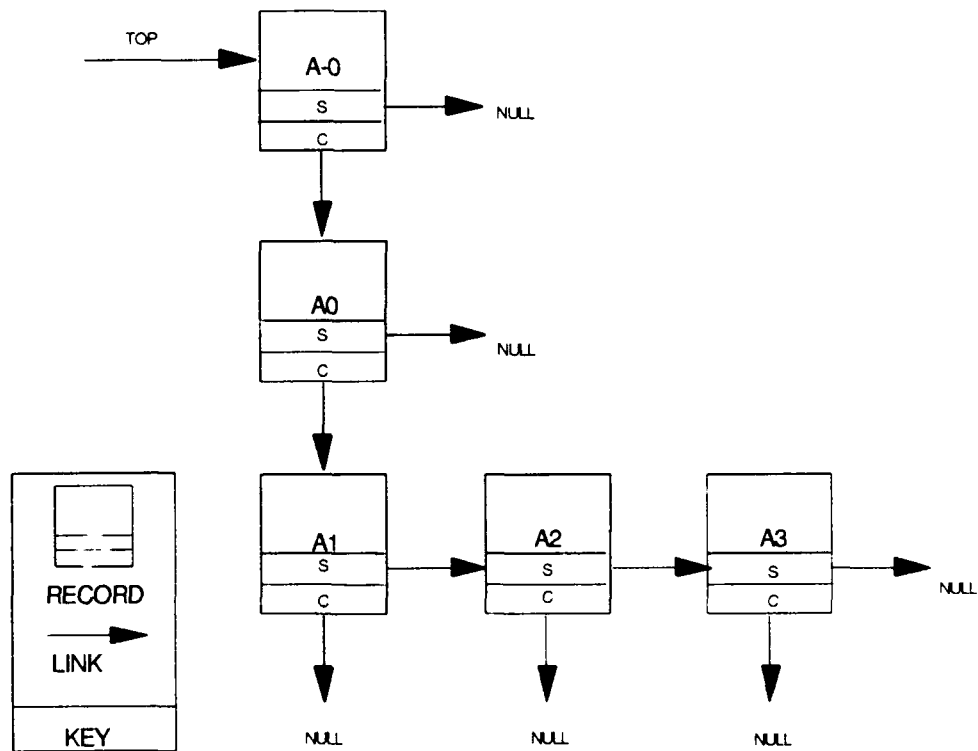


Figure 54. SADT Model represented as a Generalized List

- Advance-Iterator-To-Next-ICOM-Relation-Tuple

Using the *Activity-Class* operations, the following steps can be followed to build the tree structure, and fill in the NAME, NUMBER, CHILD, SIBLING, and PARENT data elements:

1. Reset the *Activity-Class* iterator using the *Reset-Activity-Iterator*.
2. Using the *Activity-Is-Child* and the *Advance-Iterator-To-Next-Activity*, iterate through the *Activity-Class* until the Activity that is NOT a child is located. This Activity is the root node of the hierarchy.
3. Recursively walk through the Children list of the root node building the CHILD and SIBLING pointers and records of the Activity Reader. When visiting each record in the *Activity-Class*, the *Value-of-Activity-At-Iterator* is used to retrieve the entire Activity record pointed to by the Iterator. The NAME and NUMBER information can be obtained directly from the record. The *Activity-Is-Child* operation also returns the Parent name. This can be used to set the PARENT pointer in the local data structure.

The remaining fields, INPUTS, CONTROLS, and OUTPUTS are filled in by using the operations of the *ICOM-Relation-Class*. A single iteration through the *ICOM-Relation-Class* using the Iterator operations and the *Value-of-ICOM-Relation-Tuple-At-Iterator* will suffice. As each *ICOM-Relation-Class* record is visited, the INPUTS, CONTROLS, and OUTPUTS linked lists can be built.

5.4.3.3 Operations Provided by the Activity Reader The Activity Reader needs to provide operations similar to the ones provided by each of the entities of the existing Abstract Model Manipulator. One difference to note for the Activity Reader Iterator routines is how they must proceed through the linked list of objects. The VHDL transformation process requires iterating the Activities in a Depth First Search manner.

- Clear-Activity
- Create-Activity

- Set-Activity-NAME
- Set-Activity-NUMBER
- Set-Activity-INPUTS
- Set-Activity-CONTROLS
- Set-Activity-OUTPUTS
- Set-Activity-CHILD
- Set-Activity-SIBLING
- Set-Activity-PARENT
- Value-of-Activity-At-Iterator
- Reset-Activity-Iterator
- Advance-Activity-Iterator
- Activity-Iterator-Done

5.4.4 *Generic Relationship Manager* The decision tables will be represented by the addition of a Generic Relationship Manager (GRM) to the existing Abstract Model. Blankenship's thesis explains this GRM in greater detail (9). Table 12 illustrates how a decision table is modeled. The fields represent the following:

Table 12. Generic Relationship Representation of Decision Table

Label	Type	Item 1	Relation	Item 2
Shut_Off_Furnace	Pre	Combustion_Sensor	=	Unsafe
	Post	Shut_Down	=	True
	Pre	Fuel_Flow	=	Unsafe
	Post	Shut_Down	=	True
	Pre	Combustion_Sensor	=	Safe
	Pre	Fuel_Flow	=	Safe
	Post	Shut_Down	=	False

- The LABEL field contains the name of the Activity to which the table belongs.

- The TYPE field is used to specify whether Item 1 is a Pre-condition or a Post-condition. This field could also contain entries such as No-pre or No-post, for rows or columns in the decision table that have no Pre- or Post- conditions, respectively.
- ITEM 1 contains the variable name.
- The RELATION field is used to specify the relationship the variable name has to the entry in the Item 2 field. Usually it will be the equality relationship, but other valid entries are $\neq$, $\geq$, $\leq$, $>$, and $<$.
- The ITEM 2 field contains the *value* corresponding to the relationship and the variable from Item 1.

The first TYPE entry will always be a Pre or No-Pre entry and represents the first condition row(column) of the decision table. The next Pre or No-Pre that follows a Post or No-Post represents the next condition row(column), and so on, until the GRM reaches the next LABEL (which indicates a new table) or is empty.

The following operations are required of the Generic Relationship Manager in order to perform the translation from SADT to the target language:

- Reset-Generic-Relationship-Iterator
- Value-Of-Generic-Relationship-At-Iterator
- Advance-Iterator-To-Next-Generic-Relationship
- Generic-Relationship-Iterator-Done

Note that this list does not include the operations to create the GRM or to initially enter the data into the fields. The reader is referred to Blankenship's thesis for a thorough discussion about all the operations (9).

5.5 Generation of the VHDL Source Code

This section describes transforming the information contained in the AM-Reader Object into VHDL source code. The transformation is outlined in a step-by-step process which corresponds to the steps introduced in Section 4.9. The generation of the source

code is essentially a process of creating and writing strings to a file. VHDL does not dictate any particular file naming rules, but a *.vhd* extension is a standard convention. The particulars of file management (create, open, close, etc.) are not addressed in the scope of this discussion.

The information required for generation of the VHDL source code is contained entirely in the local data structures defined in the previous sections of this chapter. The following discussion will use `DATA_ELEMENT` to refer to the local data element data structure (defined in Section 5.4.2), and `ACTIVITY` to refer to the local Activity Reader data structure (defined in Section 5.4.3).

1. Generate a package containing type definitions for the SADT model.

- (a) All the required information for generating VHDL type definitions is contained in the `DATA_ELEMENT` data structure. Generation of the types definitions requires two iterations through the local data element data structure. On the first pass through, type definitions are created for each data element where `TYPE≠RECORD`. As an example, if the following information exists for a data element:

NAME:	On_Off_Indicator
TYPE:	Enumerated
INIT:	On
CHILDREN:	
VALUES:	On, Off

then the VHDL type declaration generated would be as follows:

```
type On_Off_Indicator_Type is On_Value, Off;
```

The process for generation of such a string is described in the following steps.

- the string *type* followed by a space is generated.
- the NAME is output followed by the string *_Type* followed by a space.
- the string *is* followed by a space is generated.
- the VALUES is output.
- a semi-colon followed by a carriage return is generated.

This detailed description will not be repeated in the remaining discussion of the transformation process. A description of where the information is contained in the AM-Reader is provided. The reader is referred to the examples of the resulting VHDL source code provided in Section 4.3.

NOTE: A check must be performed of each data element name and any enumerated values against a list of the VHDL reserved words. If a reserved word is being used, append a character or string to the word to make it unique. In the example above, the string *_Value* was appended to the variable name since *On* is a VHDL reserved word. A warning to the user of the action taken should be generated.

The second iteration through the data element structure is used to build the RECORD types. As an example the data element:

```

NAME:      Dest_Button
TYPE:      Record
INIT:      0,0
CHILDREN:  Floor, Lift
VALUES:

```

would be generated in VHDL as:

```

type Dest_Button_Type is record
    Floor : Floor_Type;
    Lift  : Lift_Type;
end record;

```

2. Generate a VHDL entity declaration for each SADT activity box. The information necessary to generate this description is contained in the DATA_ELEMENT and ACTIVITY data structures.

- (a) The entity name is in ACTIVITY.NAME
- (b) Use the ACTIVITY.INPUTS and ACTIVITY.CONTROLS as the **in** ports in the entity port declaration.
- (c) Use the ACTIVITY.OUTPUTS as the **out** ports in the entity port declaration.
- (d) The type designation for each port must be retrieved by iterating the DATA_ELEMENT list until a record matching the port name is located.

As an example, the following ACTIVITY exists:

```

NAME:      Control_Water_Valve
NUMBER:    A4
INPUTS:
CONTROLS:  Water_Temp
OUTPUTS:   Water_Valve_Control
CHILD:     null
SIBLING:   null
PARENT:    Control_Heater

```

then the VHDL code generated would look like:

```

entity Control_Water_Valve is
port ( Water_Temp      : in Water_Temp_Type;
      Water_Valve_Control : out Water_Valve_Control_Type);
end Control_Water_Valve;

```

3. Generate a behavioral VHDL *Architecture Body* for each leaf-node activity. Leaf-node activities are identified by a null value in the ACTIVITY.CHILD field.
 - (a) Generate the *Declaration Statement* for the body. The ACTIVITY.NAME field provided the required information.
 - (b) Generate a *Process Block* within the Architecture Behavior. Sensitize the *Process Block* to the ACTIVITY.INPUTs, and ACTIVITY.CONTROLS of the leaf-node activity.
 - (c) Read the behavior definition for the leaf node activity from the decision tables and create the corresponding if-then-elsif construct in the *Process Block*. This operation is performed using the operations provided in the Generic Relationship Manager described in Section 5.4.4 and in (9).
4. Generate a structural VHDL *Architecture Body* for each parent SADT activity box. Parent activities have a list of their children in the ACTIVITY.CHILD field.
 - (a) Generate the *Declaration Statement* for the body. The ACTIVITY.NAME field provides the required information.
 - (b) Generate Component Declarations in the Architecture Structure for each child of the activity being defined. The children are identified by traversing the list in the ACTIVITY.CHILD field of the parent activity.
 - (c) Generate a VHDL *Signal* declaration for each non-boundary arrow present in the child activity. Non-boundary arrows can be identified by checking the child lists of Inputs, Controls, and Outputs against the corresponding parent lists. A pointer to the parent record for each child is provided in the ACTIVITY.PARENT field. For example an Input in the child ACTIVITY.INPUTS list can be checked against the ACTIVITY.PARENT.INPUTS list for its existence. If it exists in the parent list, it is a boundary arrow, otherwise it is a non-boundary arrow.
 - (d) Generate component instantiations for each child activity. The convention proposed was to use the Activity number as the instantiation name. This information is contained in the ACTIVITY.NUMBER field.
 - (e) Generate port maps for the instantiated components using the rules stated in Step 2, and the local signals generated in Step 4c. This step is logically equivalent to wiring the VHDL components together with signals.
 - (f) Determine if any Output boundary arrows were replaced in the component port maps (generated in Step 4e) with one of the local signals generated in Step 4c. If so, generate a signal assignment statement for each occurrence to assign the value of the local signal to the boundary arrow.

5. Generate a VHDL entity declaration for testbench entity.
 - (a) Use the `ACTIVITY.INPUTS` and `ACTIVITY.CONTROLS` of the corresponding A-0 entity (the UUT) as **out** ports in the testbench entity port declaration.
 - (b) Use the `ACTIVITY.OUTPUTS` of the corresponding A-0 entity (UUT) as the **in** ports in the testbench entity port declaration.
6. Generate a behavioral VHDL *Architecture Body* for the testbench entity containing the desired test behavior. This step is not automatable with the information available in the SADT Essential model.
7. Generate a VHDL entity declaration for the system entity. No information is contained in the AM-Reader for this entity, and the declaration will not have a port map. Use of the name *System* appears to be a standard naming convention for this entity.
8. Generate a VHDL structural *Architecture Body* declaration for the System entity.
 - (a) Generate the *Declaration Statement* for the body.
 - (b) Generate component declarations for the VHDL entity which corresponds to the A-0 SADT activity, and for the testbench entity.
 - (c) Generate local VHDL *Signals* for each of the testbench ports.
 - (d) Generate component instantiations for each component. Standard names like *UUT* and *Testbench* can be used for the SADT system and testbench components.
 - (e) Generate port maps for the instantiated components using the local signals created in Step 8c.
9. Generate the system *Component Configuration* declaration. This step is not addressed as being automatable with the information available in the SADT Transformer, however, a method for automatically generating this may be feasible. The description of the possible method is described as a topic for future research in Chapter VII.

5.5.1 Completeness of the Code Generation Performing the steps in the transformation process results in the generation of a VHDL specification which represents the entire hierarchical SADT specification. Automation of Steps 6 and 9 of the process were not addressed in the context of this design, however, potential methods for automating these steps are proposed as additional areas of research in Section 7.4.

After the VHDL specification is generated, the VHDL source code needs to be compiled and simulated in the VHDL environment. The next section describes some compile time errors which may occur stemming from the rigorous type-checking of VHDL. These errors will need to be corrected before the simulation can be invoked.

5.5.2 Cautions of the Automated Process Certain assumptions were made in proposing the automation of the information contained in the Abstract Model. These are restrictions based upon the implementation of the SADT subset and the VHDL subset used in this research.

1. Activity names are unique.
2. Data element names are unique.
3. The use of VHDL reserved words was avoided.
4. Direct feedback to the same activity was not used.
5. Data element type information exists for the NAME, TYPE, INIT, CHILDREN, and VALUES fields in the DATA_ELEMENT record.
6. Activity information exists for the NAME, NUMBER, INPUTS, CONTROLS, OUTPUTS, CHILD, SIBLING, and PARENT fields in the ACTIVITY record.

In addition to these assumptions, a word of caution must be stated. The VHDL is a strongly-typed language. Type checking is performed in all assignment and comparison operations. Using the automated generation of the types package may result in VHDL syntax errors due to invalid types. These 'invalid type' errors will have to be corrected manually so the VHDL model will compile. As an example, suppose the following VHDL type and signal definitions were generated:

```
type Current_Direction_Type is (Up, Down);
type Desired_Direction_Type is (Up, Down);

signal Current_Direction : Current_Direction_Type;
signal Desired_Direction : Desired_Direction_Type;
```

It might be then desired to check if `Current_Direction = Desired_Direction`. However, this is not allowed in VHDL because the two signals are of different type. The solution to this problem used in this effort was to manually create a separate type, like:

```
type Direction_Type is (Up, Down);
```

and declare the signals using this new type:

```
signal Current_Direction : Direction_Type;  
signal Desired_Direction : Direction_Type;
```

Now the condition `Current_Direction = Desired_Direction` can be checked.

5.6 Summary

This chapter provided a detailed design for the implementation of the SADT to VHDL transformation. The existing Abstract Model was used as a baseline. A data element object, an activity reader object, and a generic relationship object are created to process the transformation. The data element object is used to generate variable declarations. The activity reader object in conjunction with the generic relationship object are used to generate the source code for the VHDL Process Blocks. Specifically, the generic relationship object is processed to retrieve the decision table logic information, which becomes the heart of the VHDL Process Block. The local representation of the Abstract Model information provided an easier means to generate the remaining VHDL hierarchy.

VI. Comparison of the VHDL and Refine Implementation Methods

This chapter compares and contrasts the SADT to Refine Translation of Douglass (16) with the SADT to VHDL Translation of this thesis. The goal of both research efforts was essentially the same: Take the informal Structured Analysis and Design Technique (SADT¹), and convert it to an executable formal target language so that the behavior of a specified system can be simulated. The resulting simulation allows the designer to validate the SADT requirements specification by observing the simulation behavior. Additionally, the formal target language source code serves as an unambiguous formal basis for moving on into the design and implementation phase of the software life cycle. The target language of Douglass's thesis is Refine², which is a wide spectrum language developed and offered by Reasoning Systems, Inc. The target language of this research is the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), which is designed to specify and simulate hardware systems. Both research efforts were accomplished to determine if SADT could be mapped into a formal, executable simulation environment, and to also compare results from the two approaches. Therefore, the two research efforts were conducted in parallel and the following paragraphs highlight significant differences between the two approaches, as they relate to the various phases. It is important to note that some of the difficulties that were experienced by each research effort was in part due to the compromises necessary in keeping a parallel effort. Conducting this initial feasibility research in parallel was a good decision, however future research should follow independent paths to separate the concerns and differences between the Refine and VHDL efforts.

6.1 Developing the SADT's

The first step in the requirements validation process is performing a requirements analysis. For the purpose of this thesis, SADT was chosen as the method used to document the results of this analysis. Generation of the SADT model proceeds in the normal top-down functional decomposition approach of the SADT methodology (21). While generating

¹SADTTM is a trademark of SofTech, Inc.

²RefineTM is a registered trademark of Reasoning Systems, Inc

the SADT model, two constraints surfaced when translating to the VHDL target language. The designer must 1) Avoid implied sequential ordering, and 2) avoid direct feedback loops. These constraints are described further in the following sections.

6.1.1 Implied Sequential Ordering The VHDL simulation generated from the SADT model is a concurrent simulation of the SADT activities. Each activity is modeled in VHDL as a separate entity. The behavior of the SADT leaf node activities is specified using a VHDL process block which becomes active any time one of the inputs to the entity changes. For example, if two SADT activities named Box1 and Box2 have a common input, Input1, then the resulting VHDL code will have two entities which become active when Input1 changes. Thus, the two processes, Box1 and Box2, will be active concurrently. Note that the activation of processes in VHDL is nondeterministic, therefore, one can not determine which box is active first. If an implied sequential order, i.e. Box2 is active after Box1, was used in the SADT model, the designer will experience results from the VHDL simulation which are different from the expected results.

This same limitation does not exist in the Refine simulation of the SADT model. In fact, just the opposite situation is true. Refine (as used in the SADT to Refine translation process) simulates sequential behavior quite well, but does not directly simulate concurrent activities. Each leaf node SADT activity is modeled in Refine as a function. The implied sequential ordering in the SADT model is implemented in the Refine simulation by invoking the functions in the desired order. Those functions are only executed by an explicit function call, and only one can execute at a time.

6.1.2 Feedback Loops The SADT methodology allows the concept of feedback to be used in the modeling process. The use of feedback poses no apparent problems in the Refine environment, but feedback must be limited in SADT models which are to be transformed into the subset of VHDL used in this research effort. There are two forms of feedback in SADT.

1. Cyclic feedback where the output loops back to the input or control of the same activity.

2. Where an output loops back to an activity of higher dominance. This feedback can be:

- (a) Dataflow Feedback - the output loops back to a previous input.
- (b) Control Feedback - the output loops back to a previous control.

Examples of these feedback types are shown in Figure 55.

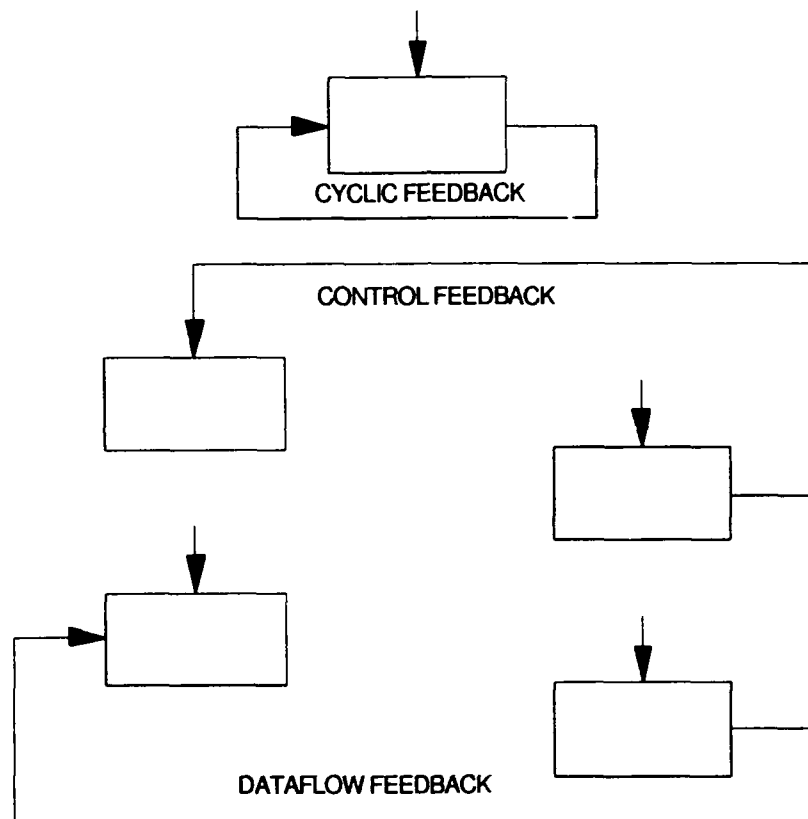


Figure 55. SADT Feedback Loop Types (32)

Cyclic feedback is allowed for an entity in the full VHDL environment if both the input and output connections to the entity are considered bi-directional. Furthermore, the internal structure of the entity must contain a *buffer*. This buffer provides the means to feedback the output to the input. The connections to the entity must then be declared to be of type INOUT. This type corresponds directly to the concept in Ada procedure

declarations where parameters can be of type IN, OUT, or INOUT. This was not done, however, in the transformation from SADT to VHDL defined in this research because the definition of the SADT methodology only uses activity connections which are IN or OUT. In addition, use of the type INOUT can easily result in a process which activates itself and never relinquishes control; i.e. an infinite loop or deadlock condition.

Dataflow and Control Feedback can be implemented in VHDL as long as the SADT model follows the following rules:

1. Each connection between activities must have a unique name.
2. The decision table (See Section 6.2) must be constructed carefully to avoid an infinite loop around the feedback path.

An example of a dataflow feedback loop used in an SADT design is shown in Figure 56.

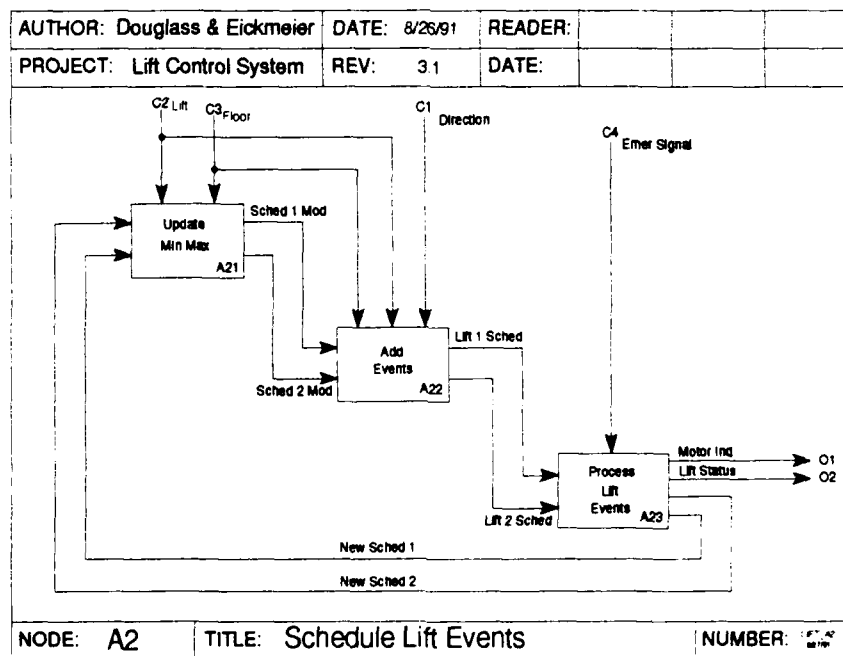


Figure 56. Lift Control System A2 Diagram using Feedback

A cyclic feedback can be redrawn as a dataflow or control feedback by adding an additional activity in the loop. Figure 57 shows how the new activity is added. This additional activity would copy the input data to the output data based upon a set of rules.

These rules would be written in a manner which avoids an infinite loop. It was discovered in the simulation of the Lift Control System that the inputs only needed to be copied to the outputs when they have changed.

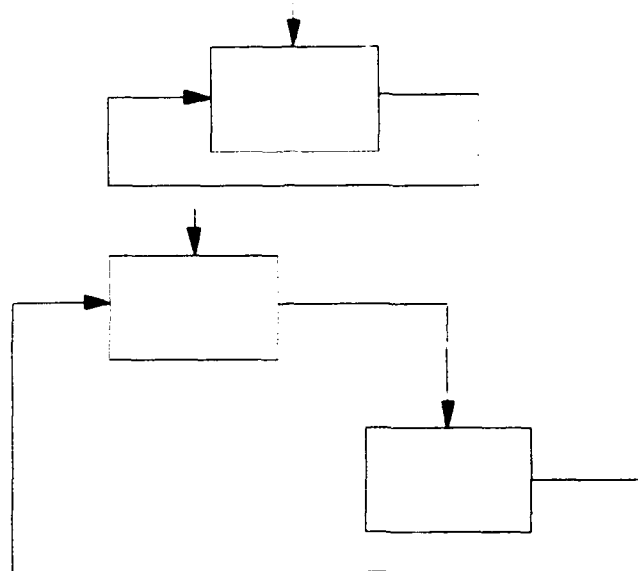


Figure 57. Cyclic Feedback Loop Redrawn

The feedback problem is avoided in the Refine transformation due to the fact that each function must be explicitly invoked in a sequential fashion. Because of this explicit function calling, the infinite loops and deadlock conditions experienced in the VHDL simulation never occur. The changes made in the Lift example problem to accommodate the VHDL transformation had no adverse impact on the Refine transformation, except that the size of the decision tables were larger.

6.2 *Creating the Decision Tables*

In order to capture detailed behavior that could later be simulated, the existing SADT model was extended by the use of decision tables. Without the decision tables, the SADT model shows that controls and inputs are transformed into outputs, but there is no formalization to describe the behavior of how the inputs are transformed, or what any of

the values might be. The decision tables give the developer the opportunity to formally indicate detailed behavior. The output values are determined by the state of the inputs and controls. A separate decision table is created for each activity of the SADT that is not further decomposed (i.e., leaf node activities). Decision Tables can be oriented such that the variables appear across the top such as in tables 13 and 14, or in the more conventional sense such that the variables appear going down the page in column 1 as shown in table 15 (34). Two areas of concern for the decision tables surfaced during the research and are covered in the following paragraphs.

6.2.1 Mutual Exclusion A decision table is mutually exclusive if each of the rows (or columns) represents a unique state of the input and control variables (36). In other words, if only one of the pre-conditions of the decision table can be true at any given time, mutual exclusion is achieved. Without mutual exclusion two (or more) conditions can be true given the same set of pre-condition values, and one of two situations exist (36, 24):

1. If the resulting post-conditions are the same, then the conditions are redundant.
2. If the resulting post-conditions are not the same, then the conditions are conflicting and lead to a contradiction. In other words, the conditions are non-deterministic.

According to Hurley's book on decision tables, a complete decision table must have no redundant rules (conditions) and no contradictory or conflicting rules (conditions) (24:10). Table 13 illustrates a table that is not mutually exclusive. Table 14 illustrates the same table that has now been made mutually exclusive. Note that on the first table, if both the Combustion Sensor and the Fuel Flow entries are Unsafe, then both the first and second rows of the decision table are true. In this particular example the post-condition for both of these rows is the same (Shut Down = True), which results in redundant conditions as mentioned above. Situations can occur where more than one pre-condition is true, but the corresponding post-conditions are different. This results in conflicting and contradictory (non-deterministic) specification of the desired post-condition. By adding the entry Combustion Sensor = Safe (see the second table) all three rows are now mutually exclusive of each other. Only one condition can be true at a given time. Thus, the decision table now

Table 13. Activity Behavior - A1 (Not Mutually Exclusive)

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
	Unsafe	True
Safe	Safe	False

Table 14. Activity Behavior - A1 (Mutually Exclusive)

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
Safe	Unsafe	True
Safe	Safe	False

Table 15. Conventionally Oriented Decision Table

Determine Status				
Emer Button.L1 Value	True	True	False	False
Emer Button.L2 Value	True	False	True	False
Motor Ind.L1 Ind				
Motor Ind.L2 Ind				
Emer Signal.L1 Value	True	True	False	False
Emer Signal.L2 Value	True	False	True	False
Current Status.L1	Out	Out	Motor Ind.L1 Ind	Motor Ind.L1 Ind
Current Status.L2	Out	Motor Ind.L2 Ind	Out	Motor Ind.L2 Ind

meets the traditional guidelines. Reduction and optimization of traditional decision tables is possible using the theory presented in (36), but was beyond the scope of this research.

6.2.1.1 Refine Implications The Refine Translation model is deterministic in nature, so ensuring that the decision tables were mutually exclusive was important. The Refine construct used to capture the decision table behavior was the Transform (38). A Transform consists of one (or a set) of pre-conditions followed by one (or a set) of post-conditions. Since the translation of a complete decision table consisted of a number of Transform statements (one for each row or column), each and every Transform whose pre-condition was valid, also made its post-condition true. In other words, more than one Transform statement could be executed if the decision table was not mutually exclusive, resulting in redundant or conflicting behavior. Therefore, forcing traditional, mutually exclusive decision tables was valid and prevented the possibility of specifying non-deterministic behavior.

6.2.1.2 VHDL Implications The VHDL Translation used an **If-Then-Elseif** structure to capture the decision table behavior. Once a valid **If** condition is found, the **Then** portion is executed and the statement is finished. Subsequent conditions are not even checked. With this type of construct, mutual exclusion is enforced during execution based upon the ordering of the rows in the table. The first true pre-condition results in the execution of the **Then** post-condition. Therefore, the decision tables need not be mutually exclusive, as long as the ordering of the rows (or columns) is watched.

The VHDL transformation could have been implemented using separate **If-Then-End If** constructs for each row in the decision table. This implementation would have mirrored the Refine implementation of Douglass (16). The implications and need for mutual exclusion described in Section 6.2.1.1 would then hold true for this alternate VHDL implementation.

Forcing traditional, mutually exclusive decision table pre-conditions was not required in the **If-Then-Elseif** approach used in this research, but would be required in the **If-Then** approach. In either case, the use of mutual exclusion offers the following benefits.

- Decision table theory (24) provides a methodical approach to creating decision tables which are complete (i.e., no states are missing).
- Creating mutually exclusive decision tables prevents the possibility of specifying non-deterministic behavior. Relying on the ordering alone to determine which pre-condition is executed can result in non-deterministic behavior if more than one true pre-condition exists in the decision table. The VHDL execution using **If-Then-Elseif** will always choose the first true pre-condition, which may not have been the intended behavior.
- Optimization and reduction techniques exist (24, 36) which provide a mechanical process for obtaining the smallest representation of behavior. Hurley (24) outlines a conversion technique which allows any mutually exclusive decision table which has no ELSE rules to be transformed into an equivalent Boolean truth table.

6.2.2 Completeness Completeness refers to the fact that all possible states of the pre-conditions are specified in the decision tables. This can be done by explicitly listing all of the combinations of pre-conditions, or by the clever use of *don't care* (blank) entries in the tables. An overall 'Else' condition can be specified in the decision tables, but it might involve some very complex pre-conditions. If a certain set of pre-conditions produces a state in which you truly don't care about the post-condition, then that state could be omitted from the decision table. The important concern that impacted both the Refine and VHDL research, was to ensure that all possible states are covered when you have a Function/Process that must result in some post-conditional change. This occurred in the *Lift Problem* when a schedule had to be copied, regardless of any other required post-condition.

6.2.3 Combinatoric Explosion One concern that applies to both the Refine and VHDL translations is the combinatoric explosion that can result from a complete decision table. As the number of variables increases, or the number of possible distinct values that a variable can have increases, the number of possible states grows at a combinatoric rate. Each one of these states must be specified and results in decision tables that are quite large. This concern is a result of using decision tables, and applies equally to both the Refine

and VHDL translations. Both implementations handle large decision tables, however they result in Refine Functions and VHDL Processes that are quite large.

6.3 Performing the Translation

In order to perform the translations to Refine and VHDL, a mapping between SADT (with the decision tables) and the target languages had to be defined. Source code can then be generated, based on the defined mappings.

6.3.1 Determining the Mappings As mentioned above, the Refine translation maps the decision table information into Transforms. VHDL maps the decision tables into an If-Then-Elself structure. Both target languages map the inputs, controls, and outputs into variables. The Refine translation recognizes the following types:

- Integer
- Real
- String
- Record

In Refine, Boolean and Enumerated types are treated as special cases of the String type. The VHDL translation recognizes the following types:

- Integer
- Real
- Enumerated
- Boolean
- Record
- Character
- String

The Refine simulation is generated by mapping the leaf node activities into Functions. The VHDL simulation is generated by mapping the entire SADT model hierarchy into a hierarchical VHDL model. This is accomplished using both VHDL Structural constructs and Behavioral constructs. Each parent activity in the SADT model is described with a VHDL structural description. This structural description specifies which and how lower level (child) activities are combined to produce the desired behavior of the parent activity. This mapping continues down the hierarchy until a leaf node activity is reached. This leaf node is then described using a VHDL behavioral description. At this level, a VHDL process block is created for each leaf node activity. This process block, which is activated by changes in the inputs and controls of the activity, contains the If-Then-Elself structure generated from the decision table.

6.3.2 Generating the Source Code Generating the Refine and VHDL source code consisted of the same two basic processes: process the variable declarations, and process the activities. Processing the variable declarations was essentially the same for both target languages. There was a significant difference in processing the activities, however. The Refine translation is only concerned with processing the Leaf Node activities. The decision table for each Leaf Node activity is translated to a Refine Function, and within each function is a series of Transforms, one for each row(column) of the decision table. VHDL on the other hand is concerned with the hierarchical nature of the SADT model. The generation of the source code entails traversing down the hierarchically structured SADT model and generating the appropriate VHDL structural or behavioral construct. Once the leaf node activity is reached, the decision table is translated in a manner similar to Refine. Each decision table is translated into a single If-Then-Elself construct. The first row(column) forms the initial If-Condition-Then-Action. Each subsequent row(column) of the decision table is translated into an Elself-Condition-Then-Action. The following example shows how the Activity Behavior given in Table 14 is transformed into both Refine and VHDL. The Refine source code is illustrated in Figure 58.

The VHDL source code is illustrated in Figure 59.


```

function Shut-Off-Furnace()=
    combustion-sensor = "unsafe" -->
        (shut-down <- "true");
    combustion-sensor = "safe" & fuel-flow = "unsafe" -->
        (shut-down <- "true");
    combustion-sensor = "safe" & fuel-flow = "safe" -->
        (shut-down <- "false")

```

Figure 58. Example of Refine Source Code

```

architecture Behavior of Shut_Off_Furnace is
begin
    process (Combustion_Sensor, Fuel_Flow) begin
        if Combustion_Sensor = UNSAFE then
            Shut_Down <= TRUE;
        elsif Fuel_Flow = UNSAFE then
            Shut_Down <= TRUE;
        elsif Combustion_Sensor = SAFE and Fuel_Flow = SAFE then
            Shut_Down <= FALSE;
        end if;
    end process;
end Behavior;

```

Figure 59. Example of VHDL Source Code

6.4 Running the Simulations

6.4.1 Generating the Test Cases Validating the behavior of the SADT model is the primary purpose of generating the Refine and VHDL simulations. To simulate the SADT model, inputs to the system are generated and the resulting outputs are compared against the expected behavior. Simulating the system behavior is very domain oriented, and a user who has domain knowledge is required to create and execute the simulations in both the Refine and VHDL environments. Table 16 gives an example test case generated to validate the behavior of the Activity shown in Table 14. (The Combustion Sensor = Unsafe causes the Shut Down variable from Table 14 to be True. This propagates through the system and results in the Expected Outputs of the overall system.)

Table 16. TEST 3 - Combustion Error while Running

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	Off
Combustion Sensor	UnSafe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Threshold		

Simulation in the Refine system consists of 1) assigning values to the system inputs, and 2) writing a test function which calls the Activity Functions in a sequential fashion. The calling sequence of the Functions is determined by the domain user. The user runs the test function and then compares the outputs generated to the expected results. The Refine system allows for great flexibility in creating and running the tests. Individual functions can be executed just as easily as entire test cases. Also, the user need not re-run an entire portion of a test just to get to a desired state. Refine allows the user to 'inject states' by changing any of the variables at any point during the simulation. Turn-around time between tests is very short, because the user only needs to re-compile the portion of the

code that actually changed. In this case, a single FORM construct that established the values of the state variables.

In the VHDL environment, simulation is performed by creating a 'Testbench.' This Testbench consists of a separate VHDL entity which exercises the SADT model under test. This Testbench is connected to the SADT model (See Figure 60), and it generates the inputs and reads the outputs of the model. The behavior of the Testbench is written by a user who is familiar with VHDL, and the problem domain. The contents of the Testbench behavior can vary greatly, but essentially it sequentially assigns values to the SADT model inputs, and then allows the simulation to run a specified amount of time. The outputs of the SADT model are then compared to the expected results. The user may also directly interface with the simulation environment. Most of the standard debugger tools are provided by the simulator. The user may trace source code files, set code breakpoints, monitor values of variables as they change, and modify the value of variables. While the capability to modify variables is present, it is not a simple task to 'inject states' during the simulation since the simulation and the variables are time dependent.

6.4.2 Compiling the Source Code The source code generated for the Refine and VHDL simulations are contained in one or more files.

To compile the source code in the Refine environment, each file is individually compiled. Those files containing declarations must be compiled first. Compiling the source code creates *.fasl* files (similar to Lisp). If only a small change has been made, then the entire file doesn't need to be re-compiled. The changed part can be re-compiled and automatically loaded without leaving the simulation environment (38).

In the VHDL environment, the source files are compiled (analyzed in VHDL terminology) in a sequential manner outside of the simulation environment. Any code unit which generates information required by another code unit must be compiled first. Any time a change is made to a source code unit, it, and all units dependent upon it, must be recompiled.

The Refine environment has distinct advantages over the VHDL environment when the code units are small (such as in the Heating System example). It allows incremental

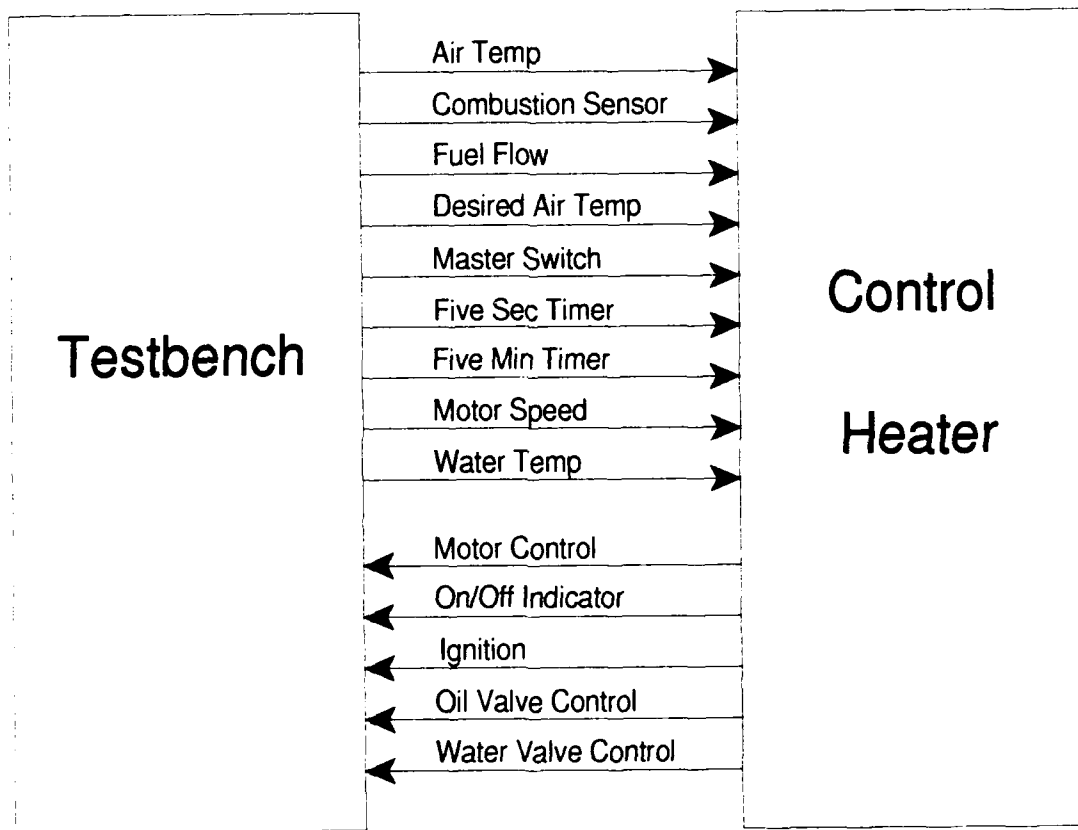


Figure 60. VHDL Testbench connection to the SADT Model

compiles to be performed without leaving the simulation environment. The VHDL environment, however, compiles much quicker when the code units become large (such as in the Lift System example problem). On identical hardware platforms, the VHDL user can leave the simulation environment, modify and compile the source code, and begin testing the changes before the Refine compile has even completed.

6.4.3 Execution of the Simulation In order to execute the Refine simulation, the .fasl files must first be loaded. Then any defined function (including the test program) can be invoked.

The VHDL simulation environment is a discrete-event simulation model based upon a simulation clock. Events are scheduled at certain clock times, and may occur instantly, or after some period of time. Therefore, executing a VHDL simulation is very time dependent. The behavior of the Testbench which exercises the SADT model must take this timing into account.

The VHDL time-based simulation has both disadvantages and advantages when compared to the Refine simulation. The addition of time adds complexity to the test environment. It also forces a particular sequence of events to occur before a system state can be reached. A Refine simulation is not dependent upon the sequential state generation. A test in Refine can set the simulation at a particular state, and proceed from there. The advantage of the VHDL time-based simulation comes from the wider range of problems which can be analyzed. A VHDL simulation can include time dependencies, and concurrent process operations. Therefore, the concurrent and time constraint aspects typical of real-time systems can be simulated and tested in VHDL.

Both the Refine and VHDL simulations can be run interactively by the user. The VHDL simulation is controlled by specifying how long the simulation runs before the next breakpoint. The Refine simulation allows the user to call individual functions, or a series of functions. In both environments, inputs can be assigned values and the output values can be evaluated. The VHDL environment provides a wide variety of debugging and monitoring features for the system. The ZYCAD VHDL simulation environment provided the capability for multiple windows to be opened. These windows could be setup to display

the changing value of variables as the simulation ran, or provide traces of the actual line of source code which was executing at the present time. The Refine environment used an EMACS split screen, with the editing on the left and the command screen on the right. This split screen was quite effective for changing something on the left (such as an input variable value), re-compiling just that change, and then immediately running a new simulation on the right.

One additional difference between the Refine and VHDL simulation is the views allowed into the original SADT model. Douglass's research implemented Refine behavior only for the leaf node level of the model, while the VHDL model of this research was explicitly constructed to be hierarchical in nature. This chosen Refine model allows analysis of the behavior only at the leaf node activity level. The chosen VHDL model can be analyzed at the leaf node level, or any other level higher in the SADT model.

6.5 Summary

Tables 17 and 18 are presented as overall summaries between the Refine and VHDL translation and simulation processes. Some of the differences between the Refine and VHDL translations were the result of specific problems encountered while maintaining a parallel effort, however those listed in the table hold in general. It is also important to note that the table reflects the particular chosen implementation of the Refine and VHDL translations.

Table 17. Similarities Between the Refine and VHDL Translations

Executable Simulation
Center Around SADT Activities
Decision Table Extension to SADT
Similar Constructs to Translate Decision Tables
Similar Data Types
Formal Basis for Design and Implementation

Table 18. Differences Between the Refine and VHDL Translations

Refine	VHDL
Models sequential operation. ^a	Models concurrent operation.
Timing is not directly modeled. ^a	Timing is directly modeled.
Simple test case generation.	Requires creation of a testbench. ^c
Execution sequence of Function calls is deterministic. ^b	Execution sequence of Function calls is non-deterministic.
Easy state injection during simulation.	Difficult state injection during simulation. ^c
Allows direct feedback in the SADT diagrams.	No direct feedback in the SADT diagrams. ^b
Transform implementation requires traditional mutually exclusive decision table pre-conditions. ^b	If-Then-Elsif implementation eliminates mandatory mutual exclusion, but ordering of pre-conditions was important. ^b
Only leaf node activities were processed — less complexity and shorter source code. ^b	Hierarchical processing of all activities. ^b
Only need to re-compile changed functions.	Re-compile changed functions and all dependencies. ^c
Slow compile time for large files. ^b	Much faster compile time for large files.

^aRefine Limitation

^bBased on Implementation Decision

^cVHDL Limitation

VII. Conclusions and Recommendations

7.1 Introduction

The objectives of this research, as stated in Chapter I, were to:

Determine an algorithm for transforming an SADT specification into a VHDL simulation; manually implement and validate the transformation; and evaluate the contribution to the software design process of simulating system specifications in VHDL.

The approach taken to achieve this goal was described in Chapters I-IV of this thesis. Chapter I provided an introduction to the problems resulting from a poorly written requirements specification. The use of simulation in VHDL was proposed as a means to validate and identify errors in the written specification. Chapter II reviewed four existing methods used to improve the requirements specification process. Chapter III defined the primary attributes of the SADT and VHDL subsets used in this research, and proposed a mapping between these subsets. Chapter IV described the process of validating the transformation defined in Chapter III through manually applying the mapping to two example problems. The Heating System, and The Lift Control System. Chapter IV concluded with a step-by-step process to generate VHDL source code from an SADT specification.

Next, Chapter V outlined how the translation process could be automated using the existing SADT Essential Model (27, 48), as modified by the extension of the Generic Relationship Manager (9). Finally, Chapter VI provided a comparison of the approach taken in this research to the approach taken by Douglass (16) where SADT is mapped to and simulated in the refine environment.

7.2 Summary of Accomplishments

As a result of the investigation done in this research effort, the following accomplishments were realized:

- A subset of VHDL was defined which supported the major features of the SADT method.

- A means to augment the SADT method, using decision tables, was developed. This allows for defining the detailed behavior of the system being specified in SADT. This detailed behavior is necessary for the generation of an executable simulation.
- An algorithm for transforming an SADT specification into a VHDL specification was developed. The resulting VHDL specification supports the entire hierarchy of the original SADT model.
- The transformation algorithm was successfully applied to two example problems: The Heating System and the Lift Control System. An executable VHDL simulation was generated. This simulation only contained the functional behavior of the Heater and Lift systems. Timing constraints were not included due to a limitation of not being able to represent timing constraints in the extended SADT model.
- The VHDL simulations of the Heater and Lift System were extended to validate the timing constraints of the problem specifications. Execution and testing of this time related behavior demonstrates the ability of VHDL to model real-time systems.
- The SADT to VHDL transformation was shown to be feasible and beneficial as a means to validate the specifications of the Heating System and Lift Control System.
- A design for automating the SADT to VHDL transformation was developed. This design shows that the existing Essential Model (27, 48), with minor modifications, can support the generation of an executable specification in VHDL. All but Steps 6 and 9 in the transformation algorithm are automatable within the design presented in this thesis. These two steps appear to be automatable with future research proposed in Section 7.4.

7.3 *Conclusions*

This research effort has investigated the feasibility and benefit of transforming an SADT model into a VHDL executable simulation. A transformation algorithm was developed and tested by manually applying the algorithm to two example problems. The transformation algorithm proved to be valid on these two problems. In fact, the resulting VHDL model mapped nearly one-to-one with the SADT model. The VHDL simulation

environment allowed viewing the behavior of the system specification at any level in the hierarchy of the model.

Execution of the VHDL simulation also proved to be beneficial in the requirements analysis process. Recall from Section 1.1 the four criteria used in validating requirements specifications (10):

- Completeness
- Consistency
- Feasibility
- Testability

The SADT specifications of the Heating System and the Lift Control System were both validated on their ability to meet these criteria in the generation and execution of the VHDL simulation.

Both completeness and consistency were tested during the definition of the decision tables which captured the desired behavior, and during the execution of the behavior. In both the Heater and the Lift, problems of incompleteness and inconsistency were identified, and had to be corrected to proceed with the simulation.

The feasibility of the specification was demonstrated through execution of the VHDL simulation. The executable specification provided a rapid-prototyping environment where behavior modifications could be made, and the results evaluated immediately.

Finally the testability of the requirement specification was determined when the test cases and the resulting Testbench definition was generated. This testing focused on generating inputs to the VHDL model of the system, and evaluating the outputs. Test cases were established for each requirement in the Heater and Lift specifications.

Validation of the requirements specification against these four criteria resulted in a better understanding of the required behavior of the system. Correcting the inconsistent and incomplete portions of the specification resulted in a better specification, and ultimately a product which better satisfies the users needs.

This section outlines several interesting topics which could be pursued as follow-on investigations in this research area.

7.4.1 Automation of the SADT to VHDL Translation Chapter V outlined a design for an automated step-by-step procedure to generating VHDL source code from the existing SADT essential model. This design could be implemented, and validated against the manually generated source code of the Heating System and the Lift Control System.

Not all of the transformation steps outlined in Section 5.5 were automatable using the extended SADT essential model proposed in this research. In particular, Step 6, the generation of a behavioral description of the Testbench, and Step 9, the generation of the system component configuration declaration, were not addressed. Additional research in these two areas may reveal methods to automate these steps. Very promising starting points for future research are:

- A table based approach similar to the decision table might be used to generate the Testbench behavior.
- Examination of the configuration declaration for the Heating System (see Appendix C, Section C.15) reveals a pattern familiar to those who have studied tree traversal algorithms. The generation of the configuration declaration starts at the root node of the SADT/VHDL model, and traverses the hierarchical tree structure in a depth first search (DFS) manner. Each time a parent node is visited, a library component corresponding to the structural architecture body is specified. When a leaf node is reached, the library component corresponding to the behavioral architecture body is specified.

The design outlined in Chapter V proposed the building of an internal representation of a tree structure in the SADT Transformer. A DFS algorithm could be implemented to traverse this tree and build the configuration declaration.

7.4.2 Additional Methods for Specifying Behavior Decision tables were used in this research effort to specify the desired behavior of the system being specified. While this

141 149

method works well for the Heating System problem, it was difficult and complex to do the same for the Lift Control System. The Lift problem was constrained to a two-lift, three-floor system. Scaling up the system with additional lifts and/or additional floors would lead to a combinatoric explosion in the decision tables. The logic and algorithms used in the SADT model and decision tables is still valid, but each new lift and/or floor added causes the number of possible states to grow at a combinatoric rate. Each additional state must be specified by adding more tables, and more conditions to each table.

New methods for dealing with this state explosion are required if this translation technique is to scale up to larger, real-world problems. Some possible options are:

- Optimization of the Decision Tables – Optimization rules for decision tables exist (34) and could be applied to reduce the number of possible states in the decision tables. In addition, Hurley (24) presents mechanical methods of transforming extended entry decision tables to limited entry decision tables, which can then be reduced through boolean logic. This boolean reduction is also a candidate for future research.
- State Transition Approach – A State transition approach could be used in the description of the System behavior. This should eliminate the redundancy required in the decision table approach to have a table column for each combination of variables. In a thesis by Miller (35), the use of state transition diagrams (STD) for specifying the behavior of large sequential circuits is described. He proposes and demonstrates the use of VHDL to verify the behavior of these circuits by transforming the state transition diagrams into VHDL behavioral specifications. The STD is represented in VHDL using a case-statement construct.
- Modeling Sequential Behavior – Section 4.8.3 pointed out the problem of modeling sequential behavior in the VHDL simulation environment as used in the SADT to VHDL transformation of this research. A possible solution of using a separate “state” process like Armstrong (1) to specify the sequence of activities was discussed. This approach has potential of being a generic solution to the problem, and warrants further research.

- Use of Specialized Design Components – In this research, the behavior of each leaf node was transformed into VHDL if-then-elsif constructs. One alternative is to create procedures/functions to describe the behavior using more of the constructs available to a VHDL programmer. (Most of the constructs available in the Ada language have been provided in VHDL.) Once created these components can be maintained in a common Design Library and re-used for different projects. For example, a common component might be a package to create and manage a Linked List.

These procedures and functions could then be used when constructing the configuration declaration. The SADT *Mechanism Arrow* could be used to specify the name of the procedure/function to use. The Mechanism Arrow was not addressed in the SADT to VHDL mapping of this research effort in order to allow for its use in this manner.

- SADT $\longrightarrow$ Refine $\longrightarrow$ VHDL – Chapter VI outlined some real advantages of using Refine to validate the non-time related behavior of a system specification. In addition, the description of the Refine environment in Section 2.2.4 described the capability of the environment to generate implementation source code from the internal Refine specification. Research could be done to investigate the benefits of transforming the original SADT specification into Refine, validating the non-time related behavior, and then having the Refine environment generate VHDL source code if time-related behavior exists in the system being validated. The VHDL simulation environment could then be used as described in this research to validate the time-related behavior.

Appendix A. The Heating System Problem

A.1 Decision Tables

Table 19. Heater Activity Behavior - A1

Shut Off Furnace		
Combustion Sensor	Fuel Flow	Shut Down
Unsafe		True
	Unsafe	True
Safe	Safe	False

Table 20. Heater Activity Behavior - A4

Control Water Circulation Valve	
Water Temp	Water Control Valve
Threshold	Open
Below Threshold	Closed

Table 21. Heater Activity Behavior - A21

Compare Temperature				
Shut Down	Master Switch	Air Temp (T_A)	Des Air Temp (T_R)	Oil Enable
True				False
	Off			False
		$\geq T_R + 2$		False
False	Heat		$> T_A + 2$	True

Table 22. Heater Activity Behavior - A22

Activate Motor			
Oil Enable	Five Min Timer	Five Sec Timer	Motor Control
True	True		On
False		True	Off

Table 23. Heater Activity Behavior - A31

Monitor Motor Speed			
Motor Control	Motor Speed	On/Off Indicator	Desired Speed
Off		Off	False
On	Below Threshold	On	False
On	Threshold	On	True

Table 24. Heater Activity Behavior - A32

Activate Ignition		
Desired Speed	Shut Down	Ignition
	True	Off
True	False	On
False	False	Off

Table 25. Heater Activity Behavior - A33

Activate Oil Valve			
Desired Speed	Shut Down	Oil Enable	Oil Control Valve
	True		Closed
False			Closed
		False	Closed
True	False	True	Open

Table 26. Test Case Key

Item	Applicable States	
Δt^a	≤ -2	≥ 2
Combustion Sensor	Safe	Unsafe
Fuel Flow	Safe	Unsafe
Master Switch	Heat	Off
Five Sec Timer	True	False
Five Min Timer	True	False
Motor Speed	Threshold	Below Threshold
Water Temp	Threshold	Below Threshold
Motor Control	On	Off
On/Off Indicator	On	Off
Ignition	On	Off
Oil Valve Control	Open	Closed
Water Valve Control	Open	Closed

^a Δt = Air Temp - Desired Air Temp

Table 27. TEST 1 - Initial Startup - Phase 1

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	On
Combustion Sensor	Safe	On/Off Indicator	On
Fuel Flow	Safe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Closed
Five Min Timer	True		
Motor Speed	Below Threshold		
Water Temperature	Below Threshold		

Table 28. TEST 1 - Initial Startup - Phase 2

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	On
Combustion Sensor	Safe	On/Off Indicator	On
Fuel Flow	Safe	Ignition	On
Master Switch	Heat	Oil Valve Control	Open
Five Sec Timer	True	Water Valve Control	Closed
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Below Threshold		

Table 29. TEST 1 - Initial Startup - Phase 3

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	On
Combustion Sensor	Safe	On/Off Indicator	On
Fuel Flow	Safe	Ignition	On
Master Switch	Heat	Oil Valve Control	Open
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Threshold		

Table 30. TEST 2 - Normal Shut Off - Phase 1

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≥ 2	Motor Control	On
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	On
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	False		
Motor Speed	Threshold		
Water Temperature	Threshold		

Table 31. TEST 2 - Normal Shut Off - Phase 2 (After 5 sec)

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≥ 2	Motor Control	Off
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Below Threshold		
Water Temperature	Threshold		

Table 32. TEST 2 - Normal Shut Off - Phase 3

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≥ 2	Motor Control	Off
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Closed
Five Min Timer	False		
Motor Speed	Below Threshold		
Water Temperature	Below Threshold		

Table 33. TEST 3 - Combustion Error while Running

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	Off
Combustion Sensor	UnSafe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Threshold		

Table 34. TEST 3 - Fuel Flow Error while Running

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	Off
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Unsafe	Ignition	Off
Master Switch	Heat	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Threshold		

Table 35. TEST 4 - Master Switch Off while Running - Phase 1 (Before 5 sec)

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	On
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	On
Master Switch	Off	Oil Valve Control	Closed
Five Sec Timer	False	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Threshold		
Water Temperature	Threshold		

Table 36. TEST 4 - Master Switch Off while Running - Phase 2 (After 5 sec)

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	Off
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Off	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Open
Five Min Timer	True		
Motor Speed	Below Threshold		
Water Temperature	Threshold		

Table 37. TEST 4 - Master Switch Off while Running - Phase 3

Given Inputs		Expected Outputs	
Item	State	Item	State
Δt	≤ -2	Motor Control	Off
Combustion Sensor	Safe	On/Off Indicator	Off
Fuel Flow	Safe	Ignition	Off
Master Switch	Off	Oil Valve Control	Closed
Five Sec Timer	True	Water Valve Control	Closed
Five Min Timer	True		
Motor Speed	Below Threshold		
Water Temperature	Below Threshold		

Appendix B. The Lift Control System Problem

B.1 Decision Tables

Table 38. Lift Activity Behavior - A11

Determine Status				
Emer Button.L1 Value	True	True	False	False
Emer Button.L2 Value	True	False	True	False
Motor Ind.L1 Ind				
Motor Ind.L2 Ind				
Emer Signal.L1 Value	True	True	False	False
Emer Signal.L2 Value	True	False	True	False
Current Status.L1	Out	Out	Motor Ind.L1 Ind	Motor Ind.L1 Ind
Current Status.L2	Out	Motor Ind.L2 Ind	Out	Motor Ind.L2 Ind

Table 39. Lift Activity Behavior - A12 Part 1

Select Lift				
Lift Status.L1 Dir				Up
Lift Status.L1 Loc				$\leq$ Sum Button.Floor
Lift Status.L2 Dir				Up
Lift Status.L2 Loc				
Emer Signal.L1 Value	True	True	False	False
Emer Signal.L2 Value	True	False	True	False
Sum Button.Dir				Up
Sum Button.Floor				
Sum Request.Dir	Nil	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir
Sum Request.Floor	0	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor
Sum Request.Lift	0	2	1	1

Table 40. Lift Activity Behavior - A12 Part 2

Select Lift				
Lift Status.L1 Dir	Up	Up	Up	Up
Lift Status.L1 Loc	> Sum Button.Floor	> Sum Button.Floor	≤ Sum Button.Floor	> Sum Button.Floor
Lift Status.L2 Dir	Up	Up	Down	Down
Lift Status.L2 Loc	≤ Sum Button.Floor	> Sum Button.Floor		
Emer Signal.L1 Value	False	False	False	False
Emer Signal.L2 Value	False	False	False	False
Sum Button.Dir	Up	Up	Up	Up
Sum Button.Floor				
Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir
Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor
Sum Request.Lift	2	1	1	2

Table 41. Lift Activity Behavior - A12 Part 3

Select Lift				
Lift Status.L1 Dir	Down	Down	Down	Up
Lift Status.L1 Loc				
Lift Status.L2 Dir	Up	Up	Down	Up
Lift Status.L2 Loc	≤ Sum Button.Floor	> Sum Button.Floor		
Emer Signal.L1 Value	False	False	False	False
Emer Signal.L2 Value	False	False	False	False
Sum Button.Dir	Up	Up	Up	Down
Sum Button.Floor				
Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir
Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor
Sum Request.Lift	2	1	1	2

Table 42. Lift Activity Behavior - A12 Part 4

Select Lift				
Lift Status.L1 Dir	Up	Up	Down	Down
Lift Status.L1 Loc			≥ Sum Button.Floor	< Sum Button.Floor
Lift Status.L2 Dir	Down	Down	Up	Up
Lift Status.L2 Loc	≥ Sum Button.Floor	< Sum Button.Floor		
Emer Signal.L1 Value	False	False	False	False
Emer Signal.L2 Value	False	False	False	False
Sum Button.Dir	Down	Down	Down	Down
Sum Button.Floor				
Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir
Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor
Sum Request.Lift	2	1	1	2

Table 43. Lift Activity Behavior - A12 Part 5

Select Lift				
Lift Status.L1 Dir	Down	Down	Down	
Lift Status.L1 Loc		$\geq$ Sum Button.Floor	$<$ Sum Button.Floor	
Lift Status.L2 Dir	Down	Down	Down	
Lift Status.L2 Loc	$\geq$ Sum Button.Floor	$<$ Sum Button.Floor	$<$ Sum Button.Floor	
Emer Signal.L1 Value	False	False	False	False
Emer Signal.L2 Value	False	False	False	False
Sum Button.Dir	Down	Down	Down	
Sum Button.Floor				
Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Sum Button.Dir	Nil
Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	Sum Button.Floor	0
Sum Request.Lift	2	1	2	0

Table 44. Lift Activity Behavior - A13

Issue Sequence Commands			
Dest Button.Lift			
Dest Button.Floor	$\neq 0$	0	0
Sum Request.Dir			
Sum Request.Floor	0	$\neq 0$	0
Sum Request.Lift			
Floor	Dest Button.Floor	Sum Request.Floor	0
Lift	Dest Button.Lift	Sum Request.Lift	0
Direction	Nil	Sum Request.Dir	Nil

Table 45. Lift Activity Behavior - A141 Part 1

Illuminate Lights			
Emer Signal.L1 Value	True	True	False
Emer Signal.L2 Value	True	False	True
Sum Button.Direction			
Sum Button.Floor	0	0	0
Dest Button.Lift			
Dest Button.Floor	$\neq 0$	$\neq 0$	$\neq 0$
Light On.Sum Fl			
Light On.Sum Dir			
Light On.Dest Lift	Dest Button.Lift	Dest Button.Lift	Dest Button.Lift
Light On.Dest Fl	Dest Button.Floor	Dest Button.Floor	Dest Button.Floor

Table 46. Lift Activity Behavior - A141 Part 2

Illuminate Lights			
Emer Signal.L1 Value	True	True	False
Emer Signal.L2 Value	True	False	True
Sum Button.Direction			
Sum Button.Floor	$\neq 0$	$\neq 0$	$\neq 0$
Dest Button.Lift			
Dest Button.Floor	0	0	0
Light On.Sum Fl		Sum Button.Floor	Sum Button.Floor
Light On.Sum Dir		Sum Button.Direction	Sum Button.Direction
Light On.Dest Lift			
Light On.Dest Fl			

Table 47. Lift Activity Behavior - A141 Part 3

Illuminate Lights			
Emer Signal.L1 Value		False	False
Emer Signal.L2 Value		False	False
Sum Button.Direction			
Sum Button.Floor	0	0	$\neq 0$
Dest Button.Lift			
Dest Button.Floor	0	$\neq 0$	0
Light On.Sum Fl	0		Sum Button.Floor
Light On.Sum Dir	Nil		Sum Button.Direction
Light On.Dest Lift	0	Dest Button.Lift	
Light On.Dest Fl	0	Dest Button.Floor	

Table 48. Lift Activity Behavior - A142

Extinguish Lights				
Lift Status.L1 Dir				
Lift Status.L1 Loc				
Lift Status.L2 Dir				
Lift Status.L2 Loc				
Current Status.L1	Stop	$\neq$ Stop	Stop	Idle
Current Status.L2	$\neq$ Stop	Stop	Stop	Idle
Sum Off.L1 Floor	Lift Status.L1 Loc		Lift Status.L1 Loc	0
Sum Off.L1 Dir	Lift Status.L1 Dir		Lift Status.L1 Dir	Nil
Dest Off.L1 Lift	1		1	0
Dest Off.L1 Floor	Lift Status.L1 Loc		Lift Status.L1 Loc	0
Sum Off.L2 Floor		Lift Status.L2 Loc	Lift Status.L2 Loc	0
Sum Off.L2 Dir		Lift Status.L2 Dir	Lift Status.L2 Dir	Nil
Dest Off.L2 Lift		2	2	0
Dest Off.L2 Floor		Lift Status.L2 Loc	Lift Status.L2 Loc	0

Table 49. Lift Activity Behavior - A21 Part 1

Update Min/Max			
New Sched 1.FL 1 Stop			
New Sched 1.FL 1 Dest			
New Sched 1.FL 2 Stop			
New Sched 1.FL 2 Dest			
New Sched 1.FL 3 Stop			
New Sched 1.FL 3 Dest			
New Sched 1.FL 1 Up			
New Sched 1.FL 2 Up			
New Sched 1.FL 2 Down			
New Sched 1.FL 3 Down			
New Sched 1.Max			
New Sched 1.Min			
New Sched 2.FL 1 Stop			
New Sched 2.FL 1 Dest			
New Sched 2.FL 2 Stop			
New Sched 2.FL 2 Dest			
New Sched 2.FL 3 Stop			
New Sched 2.FL 3 Dest			
New Sched 2.FL 1 Up			
New Sched 2.FL 2 Up			
New Sched 2.FL 2 Down			
New Sched 2.FL 3 Down			
New Sched 2.Max			
New Sched 2.Min			
Floor	0	$\geq$ New Sched 1.Max	$\leq$ New Sched 1.Min
Floor	0	$>$ New Sched 1.Min	$<$ New Sched 1.Max
Lift	0	1	1
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	New Sched 1.Max	Floor	New Sched 1.Max
Sched 1 Mod.Min	New Sched 1.Min	New Sched 1.Min	Floor
Sched 2 Mod.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop
Sched 2 Mod.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest
Sched 2 Mod.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop
Sched 2 Mod.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest
Sched 2 Mod.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop
Sched 2 Mod.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest
Sched 2 Mod.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up
Sched 2 Mod.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up
Sched 2 Mod.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down
Sched 2 Mod.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down
Sched 2 Mod.Max	New Sched 2.Max	New Sched 2.Max	New Sched 2.Max
Sched 2 Mod.Min	New Sched 2.Max	New Sched 2.Max	New Sched 2.Max

Table 50. Lift Activity Behavior - A21 Part 2

Update Min/Max		
New Sched 1.FL 1 Stop		
New Sched 1.FL 1 Dest		
New Sched 1.FL 2 Stop		
New Sched 1.FL 2 Dest		
New Sched 1.FL 3 Stop		
New Sched 1.FL 3 Dest		
New Sched 1.FL 1 Up		
New Sched 1.FL 2 Up		
New Sched 1.FL 2 Down		
New Sched 1.FL 3 Down		
New Sched 1.Max		
New Sched 1.Min		
New Sched 2.FL 1 Stop		
New Sched 2.FL 1 Dest		
New Sched 2.FL 2 Stop		
New Sched 2.FL 2 Dest		
New Sched 2.FL 3 Stop		
New Sched 2.FL 3 Dest		
New Sched 2.FL 1 Up		
New Sched 2.FL 2 Up		
New Sched 2.FL 2 Down		
New Sched 2.FL 3 Down		
New Sched 2.Max		
New Sched 2.Min		
Floor	> New Sched 2.Max	< New Sched 2.Min
Floor	> New Sched 2.Min	< New Sched 2.Max
Lift	2	2
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	New Sched 1.Max	New Sched 1.Max
Sched 1 Mod.Min	New Sched 1.Min	New Sched 1.Min
Sched 2 Mod.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop
Sched 2 Mod.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest
Sched 2 Mod.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop
Sched 2 Mod.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest
Sched 2 Mod.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop
Sched 2 Mod.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest
Sched 2 Mod.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up
Sched 2 Mod.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up
Sched 2 Mod.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down
Sched 2 Mod.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down
Sched 2 Mod.Max	Floor	New Sched 2.Max
Sched 2 Mod.Min	New Sched 2.Min	Floor

Table 51. Lift Activity Behavior - A21 Part 3

Update Min/Max		
New Sched 1.FL 1 Stop		
New Sched 1.FL 1 Dest		
New Sched 1.FL 2 Stop		
New Sched 1.FL 2 Dest		
New Sched 1.FL 3 Stop		
New Sched 1.FL 3 Dest		
New Sched 1.FL 1 Up		
New Sched 1.FL 2 Up		
New Sched 1.FL 2 Down		
New Sched 1.FL 3 Down		
New Sched 1.Max		
New Sched 1.Min		
New Sched 2.FL 1 Stop		
New Sched 2.FL 1 Dest		
New Sched 2.FL 2 Stop		
New Sched 2.FL 2 Dest		
New Sched 2.FL 3 Stop		
New Sched 2.FL 3 Dest		
New Sched 2.FL 1 Up		
New Sched 2.FL 2 Up		
New Sched 2.FL 2 Down		
New Sched 2.FL 3 Down		
New Sched 2.Max		
New Sched 2.Min		
Floor	< New Sched 1.Min	< New Sched 2.Min
Floor	≥ New Sched 1.Max	≥ New Sched 2.Max
Lift	1	2
Sched 1 Mod.FL 1 Stop	New Sched 1.FL 1 Stop	New Sched 1.FL 1 Stop
Sched 1 Mod.FL 1 Dest	New Sched 1.FL 1 Dest	New Sched 1.FL 1 Dest
Sched 1 Mod.FL 2 Stop	New Sched 1.FL 2 Stop	New Sched 1.FL 2 Stop
Sched 1 Mod.FL 2 Dest	New Sched 1.FL 2 Dest	New Sched 1.FL 2 Dest
Sched 1 Mod.FL 3 Stop	New Sched 1.FL 3 Stop	New Sched 1.FL 3 Stop
Sched 1 Mod.FL 3 Dest	New Sched 1.FL 3 Dest	New Sched 1.FL 3 Dest
Sched 1 Mod.FL 1 Up	New Sched 1.FL 1 Up	New Sched 1.FL 1 Up
Sched 1 Mod.FL 2 Up	New Sched 1.FL 2 Up	New Sched 1.FL 2 Up
Sched 1 Mod.FL 2 Down	New Sched 1.FL 2 Down	New Sched 1.FL 2 Down
Sched 1 Mod.FL 3 Down	New Sched 1.FL 3 Down	New Sched 1.FL 3 Down
Sched 1 Mod.Max	Floor	New Sched 1.Max
Sched 1 Mod.Min	Floor	New Sched 1.Min
Sched 2 Mod.FL 1 Stop	New Sched 2.FL 1 Stop	New Sched 2.FL 1 Stop
Sched 2 Mod.FL 1 Dest	New Sched 2.FL 1 Dest	New Sched 2.FL 1 Dest
Sched 2 Mod.FL 2 Stop	New Sched 2.FL 2 Stop	New Sched 2.FL 2 Stop
Sched 2 Mod.FL 2 Dest	New Sched 2.FL 2 Dest	New Sched 2.FL 2 Dest
Sched 2 Mod.FL 3 Stop	New Sched 2.FL 3 Stop	New Sched 2.FL 3 Stop
Sched 2 Mod.FL 3 Dest	New Sched 2.FL 3 Dest	New Sched 2.FL 3 Dest
Sched 2 Mod.FL 1 Up	New Sched 2.FL 1 Up	New Sched 2.FL 1 Up
Sched 2 Mod.FL 2 Up	New Sched 2.FL 2 Up	New Sched 2.FL 2 Up
Sched 2 Mod.FL 2 Down	New Sched 2.FL 2 Down	New Sched 2.FL 2 Down
Sched 2 Mod.FL 3 Down	New Sched 2.FL 3 Down	New Sched 2.FL 3 Down
Sched 2 Mod.Max	New Sched 2.Max	Floor
Sched 2 Mod.Min	New Sched 2.Min	Floor

Table 52. Lift Activity Behavior - A22 - Part 1

Add Events			
Floor	1	2	3
Lift	1	1	1
Direction	Nil	Nil	Nil
Sched 1 Mod.FL 1 Stop			
Sched 1 Mod.FL 1 Dest			
Sched 1 Mod.FL 2 Stop			
Sched 1 Mod.FL 2 Dest			
Sched 1 Mod.FL 3 Stop			
Sched 1 Mod.FL 3 Dest			
Sched 1 Mod.FL 1 Up			
Sched 1 Mod.FL 2 Up			
Sched 1 Mod.FL 2 Down			
Sched 1 Mod.FL 3 Down			
Sched 1 Mod.Max			
Sched 1 Mod.Min			
Sched 2 Mod.FL 1 Stop			
Sched 2 Mod.FL 1 Dest			
Sched 2 Mod.FL 2 Stop			
Sched 2 Mod.FL 2 Dest			
Sched 2 Mod.FL 3 Stop			
Sched 2 Mod.FL 3 Dest			
Sched 2 Mod.FL 1 Up			
Sched 2 Mod.FL 2 Up			
Sched 2 Mod.FL 2 Down			
Sched 2 Mod.FL 3 Down			
Sched 2 Mod.Max			
Sched 2 Mod.Min			
Lift 1 Sched.FL 1 Stop	True	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop
Lift 1 Sched.FL 1 Dest	True	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest
Lift 1 Sched.FL 2 Stop	Sched 1 Mod.FL 2 Stop	True	Sched 1 Mod.FL 2 Stop
Lift 1 Sched.FL 2 Dest	Sched 1 Mod.FL 2 Dest	True	Sched 1 Mod.FL 2 Dest
Lift 1 Sched.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop	True
Lift 1 Sched.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest	True
Lift 1 Sched.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up
Lift 1 Sched.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up
Lift 1 Sched.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down
Lift 1 Sched.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down
Lift 1 Sched.Max	Sched 1 Mod.Max	Sched 1 Mod.Max	Sched 1 Mod.Max
Lift 1 Sched.Min	Sched 1 Mod.Min	Sched 1 Mod.Min	Sched 1 Mod.Min
Lift 2 Sched.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop
Lift 2 Sched.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest
Lift 2 Sched.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop
Lift 2 Sched.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest
Lift 2 Sched.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop
Lift 2 Sched.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest
Lift 2 Sched.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up
Lift 2 Sched.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up
Lift 2 Sched.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down
Lift 2 Sched.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down
Lift 2 Sched.Max	Sched 2 Mod.Max	Sched 2 Mod.Max	Sched 2 Mod.Max
Lift 2 Sched.Min	Sched 2 Mod.Min	Sched 2 Mod.Min	Sched 2 Mod.Min

Table 53. Lift Activity Behavior - A22 - Part 2

Add Events			
Floor	1	2	2
Lift	1	1	1
Direction	Up	Up	Down
Sched 1 Mod.FL 1 Stop			
Sched 1 Mod.FL 1 Dest			
Sched 1 Mod.FL 2 Stop			
Sched 1 Mod.FL 2 Dest			
Sched 1 Mod.FL 3 Stop			
Sched 1 Mod.FL 3 Dest			
Sched 1 Mod.FL 1 Up			
Sched 1 Mod.FL 2 Up			
Sched 1 Mod.FL 2 Down			
Sched 1 Mod.FL 3 Down			
Sched 1 Mod.Max			
Sched 1 Mod.Min			
Sched 2 Mod.FL 1 Stop			
Sched 2 Mod.FL 1 Dest			
Sched 2 Mod.FL 2 Stop			
Sched 2 Mod.FL 2 Dest			
Sched 2 Mod.FL 3 Stop			
Sched 2 Mod.FL 3 Dest			
Sched 2 Mod.FL 1 Up			
Sched 2 Mod.FL 2 Up			
Sched 2 Mod.FL 2 Down			
Sched 2 Mod.FL 3 Down			
Sched 2 Mod.Max			
Sched 2 Mod.Min			
Lift 1 Sched.FL 1 Stop	True	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop
Lift 1 Sched.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest
Lift 1 Sched.FL 2 Stop	Sched 1 Mod.FL 2 Stop	True	True
Lift 1 Sched.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest
Lift 1 Sched.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop
Lift 1 Sched.FL 3 Des	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest
Lift 1 Sched.FL 1 Up	True	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up
Lift 1 Sched.FL 2 Up	Sched 1 Mod.FL 2 Up	True	Sched 1 Mod.FL 2 Up
Lift 1 Sched.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down	True
Lift 1 Sched.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down
Lift 1 Sched.Max	Sched 1 Mod.Max	Sched 1 Mod.Max	Sched 1 Mod.Max
Lift 1 Sched.Min	Sched 1 Mod.Min	Sched 1 Mod.Min	Sched 1 Mod.Min
Lift 2 Sched.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop
Lift 2 Sched.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest
Lift 2 Sched.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop
Lift 2 Sched.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest
Lift 2 Sched.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop
Lift 2 Sched.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest
Lift 2 Sched.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up
Lift 2 Sched.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up
Lift 2 Sched.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down
Lift 2 Sched.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down
Lift 2 Sched.Max	Sched 2 Mod.Max	Sched 2 Mod.Max	Sched 2 Mod.Max
Lift 2 Sched.Min	Sched 2 Mod.Min	Sched 2 Mod.Min	Sched 2 Mod.Min

Table 54. Lift Activity Behavior - A22 - Part 3

Add Events			
Floor	3	1	2
Lift	1	2	2
Direction	Down	Nil	Nil
Sched 1 Mod.FL 1 Stop			
Sched 1 Mod.FL 1 Dest			
Sched 1 Mod.FL 2 Stop			
Sched 1 Mod.FL 2 Dest			
Sched 1 Mod.FL 3 Stop			
Sched 1 Mod.FL 3 Dest			
Sched 1 Mod.FL 1 Up			
Sched 1 Mod.FL 2 Up			
Sched 1 Mod.FL 2 Down			
Sched 1 Mod.FL 3 Down			
Sched 1 Mod.Max			
Sched 1 Mod.Min			
Sched 2 Mod.FL 1 Stop			
Sched 2 Mod.FL 1 Dest			
Sched 2 Mod.FL 2 Stop			
Sched 2 Mod.FL 2 Dest			
Sched 2 Mod.FL 3 Stop			
Sched 2 Mod.FL 3 Dest			
Sched 2 Mod.FL 1 Up			
Sched 2 Mod.FL 2 Up			
Sched 2 Mod.FL 2 Down			
Sched 2 Mod.FL 3 Down			
Sched 2 Mod.Max			
Sched 2 Mod.Min			
Lift 1 Sched.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop
Lift 1 Sched.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest
Lift 1 Sched.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop
Lift 1 Sched.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest
Lift 1 Sched.FL 3 Stop	True	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop
Lift 1 Sched.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest
Lift 1 Sched.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up
Lift 1 Sched.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up
Lift 1 Sched.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down
Lift 1 Sched.FL 3 Down	True	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down
Lift 1 Sched.Max	Sched 1 Mod.Max	Sched 1 Mod.Max	Sched 1 Mod.Max
Lift 1 Sched.Min	Sched 1 Mod.Min	Sched 1 Mod.Min	Sched 1 Mod.Min
Lift 2 Sched.FL 1 Stop	Sched 2 Mod.FL 1 Stop	True	Sched 2 Mod.FL 1 Stop
Lift 2 Sched.FL 1 Dest	Sched 2 Mod.FL 1 Dest	True	Sched 2 Mod.FL 1 Dest
Lift 2 Sched.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop	True
Lift 2 Sched.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest	True
Lift 2 Sched.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop
Lift 2 Sched.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest
Lift 2 Sched.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up
Lift 2 Sched.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up
Lift 2 Sched.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down
Lift 2 Sched.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down
Lift 2 Sched.Max	Sched 2 Mod.Max	Sched 2 Mod.Max	Sched 2 Mod.Max
Lift 2 Sched.Min	Sched 2 Mod.Min	Sched 2 Mod.Min	Sched 2 Mod.Min

Table 55. Lift Activity Behavior - A22 - Part 4

Add Events			
Floor	3	1	2
Lift	2	2	2
Direction	Nil	Up	Up
Sched 1 Mod.FL 1 Stop			
Sched 1 Mod.FL 1 Dest			
Sched 1 Mod.FL 2 Stop			
Sched 1 Mod.FL 2 Dest			
Sched 1 Mod.FL 3 Stop			
Sched 1 Mod.FL 3 Dest			
Sched 1 Mod.FL 1 Up			
Sched 1 Mod.FL 2 Up			
Sched 1 Mod.FL 2 Down			
Sched 1 Mod.FL 3 Down			
Sched 1 Mod.Max			
Sched 1 Mod.Min			
Sched 2 Mod.FL 1 Stop			
Sched 2 Mod.FL 1 Dest			
Sched 2 Mod.FL 2 Stop			
Sched 2 Mod.FL 2 Dest			
Sched 2 Mod.FL 3 Stop			
Sched 2 Mod.FL 3 Dest			
Sched 2 Mod.FL 1 Up			
Sched 2 Mod.FL 2 Up			
Sched 2 Mod.FL 2 Down			
Sched 2 Mod.FL 3 Down			
Sched 2 Mod.Max			
Sched 2 Mod.Min			
Lift 1 Sched.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop
Lift 1 Sched.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest
Lift 1 Sched.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop
Lift 1 Sched.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest
Lift 1 Sched.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop
Lift 1 Sched.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest
Lift 1 Sched.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up
Lift 1 Sched.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up
Lift 1 Sched.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down
Lift 1 Sched.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down
Lift 1 Sched.Max	Sched 1 Mod.Max	Sched 1 Mod.Max	Sched 1 Mod.Max
Lift 1 Sched.Min	Sched 1 Mod.Min	Sched 1 Mod.Min	Sched 1 Mod.Min
Lift 2 Sched.FL 1 Stop	Sched 2 Mod.FL 1 Stop	True	Sched 2 Mod.FL 1 Stop
Lift 2 Sched.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest
Lift 2 Sched.FL 2 Stop	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop	True
Lift 2 Sched.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest
Lift 2 Sched.FL 3 Stop	True	Sched 2 Mod.FL 3 Stop	Sched 2 Mod.FL 3 Stop
Lift 2 Sched.FL 3 Dest	True	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest
Lift 2 Sched.FL 1 Up	Sched 2 Mod.FL 1 Up	True	Sched 2 Mod.FL 1 Up
Lift 2 Sched.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up	True
Lift 2 Sched.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down
Lift 2 Sched.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down	Sched 2 Mod.FL 3 Down
Lift 2 Sched.Max	Sched 2 Mod.Max	Sched 2 Mod.Max	Sched 2 Mod.Max
Lift 2 Sched.Min	Sched 2 Mod.Min	Sched 2 Mod.Min	Sched 2 Mod.Min

Table 56. Lift Activity Behavior - A22 - Part 5

Add Events			
Floor	2	3	0
Lift	2	2	0
Direction	Down	Down	Nil
Sched 1 Mod.FL 1 Stop			
Sched 1 Mod.FL 1 Dest			
Sched 1 Mod.FL 2 Stop			
Sched 1 Mod.FL 2 Dest			
Sched 1 Mod.FL 3 Stop			
Sched 1 Mod.FL 3 Dest			
Sched 1 Mod.FL 1 Up			
Sched 1 Mod.FL 2 Up			
Sched 1 Mod.FL 2 Down			
Sched 1 Mod.FL 3 Down			
Sched 1 Mod.Max			
Sched 1 Mod.Min			
Sched 2 Mod.FL 1 Stop			
Sched 2 Mod.FL 1 Dest			
Sched 2 Mod.FL 2 Stop			
Sched 2 Mod.FL 2 Dest			
Sched 2 Mod.FL 3 Stop			
Sched 2 Mod.FL 3 Dest			
Sched 2 Mod.FL 1 Up			
Sched 2 Mod.FL 2 Up			
Sched 2 Mod.FL 2 Down			
Sched 2 Mod.FL 3 Down			
Sched 2 Mod.Max			
Sched 2 Mod.Min			
Lift 1 Sched.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop	Sched 1 Mod.FL 1 Stop
Lift 1 Sched.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest	Sched 1 Mod.FL 1 Dest
Lift 1 Sched.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop	Sched 1 Mod.FL 2 Stop
Lift 1 Sched.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest	Sched 1 Mod.FL 2 Dest
Lift 1 Sched.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop	Sched 1 Mod.FL 3 Stop
Lift 1 Sched.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest	Sched 1 Mod.FL 3 Dest
Lift 1 Sched.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up	Sched 1 Mod.FL 1 Up
Lift 1 Sched.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up	Sched 1 Mod.FL 2 Up
Lift 1 Sched.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down	Sched 1 Mod.FL 2 Down
Lift 1 Sched.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down	Sched 1 Mod.FL 3 Down
Lift 1 Sched.Max	Sched 1 Mod.Max	Sched 1 Mod.Max	Sched 1 Mod.Max
Lift 1 Sched.Min	Sched 1 Mod.Min	Sched 1 Mod.Min	Sched 1 Mod.Min
Lift 2 Sched.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop	Sched 2 Mod.FL 1 Stop
Lift 2 Sched.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest	Sched 2 Mod.FL 1 Dest
Lift 2 Sched.FL 2 Stop	True	Sched 2 Mod.FL 2 Stop	Sched 2 Mod.FL 2 Stop
Lift 2 Sched.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest	Sched 2 Mod.FL 2 Dest
Lift 2 Sched.FL 3 Stop	Sched 2 Mod.FL 3 Stop	True	Sched 2 Mod.FL 3 Stop
Lift 2 Sched.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest	Sched 2 Mod.FL 3 Dest
Lift 2 Sched.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up	Sched 2 Mod.FL 1 Up
Lift 2 Sched.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up	Sched 2 Mod.FL 2 Up
Lift 2 Sched.FL 2 Down	True	Sched 2 Mod.FL 2 Down	Sched 2 Mod.FL 2 Down
Lift 2 Sched.FL 3 Down	Sched 2 Mod.FL 3 Down	True	Sched 2 Mod.FL 3 Down
Lift 2 Sched.Max	Sched 2 Mod.Max	Sched 2 Mod.Max	Sched 2 Mod.Max
Lift 2 Sched.Min	Sched 2 Mod.Min	Sched 2 Mod.Min	Sched 2 Mod.Min

Table 57. Lift Activity Behavior - A231 Part 1

Process Lift 1				
Emer Signal.L1 Value	True	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop				
Lift 1 Sched.F12 Dest				
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up				
Lift 1 Sched.F12 Down		False		
Lift 1 Sched.F13 Down		False		
Lift 1 Sched.Max			0	> Lift Status.L1 Loc
Lift 1 Sched.Min		≠ Lift 1 Sched.Max	4	
Motor Ind.L1 Ind		Run		Stop
Lift Status.L1 Dir		Down		Up
Lift Status.L1 Loc		1		1
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	False	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	False	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	False	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max
New Sched 1.Min	Lift 1 Sched.Min	4	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Stop	Stop	Idle	Run
L1 Status.Dir	Lift Status.L1 Dir	Up	Lift Status.L1 Dir	Up
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc + 1

Table 58. Lift Activity Behavior - A231 Part 2

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	False	True		True
Lift 1 Sched.F12 Dest		True		True
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up		False		False
Lift 1 Sched.F12 Down		False		False
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max		> Lift Status.L1 Loc	> Lift Status.L1 Loc	= Lift Status.L1 Loc
Lift 1 Sched.Min				≠ Lift 1 Sched.Max
Motor Ind.L1 Ind	Run	Run	Stop	
Lift Status.L1 Dir	Up	Up	Up	Up
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	False	Lift 1 Sched.F12 Dest	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	0
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Run	Stop	Run	Stop
L1 Status.Dir	Up	Up	Up	Down
L1 Status.Loc	Lift Status.L1 Loc + 1	Lift Status.L1 Loc	Lift Status.L1 Loc + 1	Lift Status.L1 Loc

Table 59. Lift Activity Behavior - A231 Part 3

Process Lift 1				
Emer Signal L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop		True	False	True
Lift 1 Sched.F12 Dest		True		True
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up		False		False
Lift 1 Sched.F12 Down		False		False
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max				≠ Lift 1 Sched.Min
Lift 1 Sched.Min	< Lift Status.L1 Loc	< Lift Status.L1 Loc		= Lift Status.L1 Loc
Motor Ind.L1 Ind	Stop	Run	Run	
Lift Status.L1 Dir	Down	Down	Down	Down
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	False	Lift 1 Sched.F12 Dest	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	4
L1 Status.Motor	Run	Stop	Run	Stop
L1 Status.Dir	Down	Down	Down	Up
L1 Status.Loc	Lift Status.L1 Loc - 1	Lift Status.L1 Loc	Lift Status.L1 Loc - 1	Lift Status.L1 Loc

Table 60. Lift Activity Behavior - A231 Part 4

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop			True	True
Lift 1 Sched.F12 Dest				
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up	False			
Lift 1 Sched.F12 Up	False		True	True
Lift 1 Sched.F12 Down			False	True
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	≠ Lift 1 Sched.Min		> Lift Status.L1 Loc	> Lift Status.L1 Loc
Lift 1 Sched.Min		< Lift Status.L1 Loc		
Motor Ind.L1 Ind	Run	Stop	Run	Run
Lift Status.L1 Dir	Up	Down	Up	Up
Lift Status.L1 Loc	3	3	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest	False	False
New Sched 1.F13 Stop	False	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	False	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	False	False
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	False	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	0	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Stop	Run	Stop	Stop
L1 Status.Dir	Down	Down	Up	Up
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc - 1	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 61. Lift Activity Behavior - A231 Part 5

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	True	True	True	True
Lift 1 Sched.F12 Dest	False			
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up	False	True	True	False
Lift 1 Sched.F12 Down	True	False	True	True
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	> Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift Status.L1 Loc
Lift 1 Sched.Min		≠ Lift 1 Sched.Max	≠ Lift 1 Sched.Max	≠ Lift 1 Sched.Max
Motor Ind.L1 Ind	Run			
Lift Status.L1 Dir	Up	Up	Up	Up
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	False	False	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	False	False	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	False
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	0	0	0
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Motor Ind.L1 Ind	Stop	Stop	Stop
L1 Status.Dir	Lift Status.L1 Dir	Up	Up	Down
L1 Status.Loc	Lift Status.L1 Loc + 1	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 62. Lift Activity Behavior - A231 Part 6

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	True	True	True	True
Lift 1 Sched.F12 Dest			False	
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up	False	True	True	False
Lift 1 Sched.F12 Down	True	True	False	True
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max				≠ Lift 1 Sched Min
Lift 1 Sched.Min	< Lift Status.L1 Loc	< Lift Status.L1 Loc	< Lift Status.L1 Loc	= Lift Status.L1 Loc
Motor Ind.L1 Ind	Run	Run	Run	
Lift Status.L1 Dir	Down	Down	Down	Down
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	False	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	False	False	Lift 1 Sched.F12 Dest	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	False	False	Lift 1 Sched.F12 Down	False
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	4
L1 Status.Motor	Stop	Stop	Motor Ind L1 Ind	Stop
L1 Status.Dir	Down	Down	Lift Status.L1 Dir	Down
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc - 1	Lift Status.L1 Loc

Table 63. Lift Activity Behavior - A231 Part 7

Process Lift 1				
Emer Signal L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	True	True	True	True
Lift 1 Sched.F12 Dest			True	True
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up	True	True	False	True
Lift 1 Sched.F12 Down	True	False	True	False
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	≠ Lift 1 Sched.Min	≠ Lift 1 Sched.Min	> Lift Status.L1 Loc	
Lift 1 Sched.Min	= Lift Status.L1 Loc	= Lift Status.L1 Loc		< Lift Status.L1 Loc
Motor Ind.L1 Ind			Run	Run
Lift Status.L1 Dir	Down	Down	Up	Down
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop
New Sched 1.F12 Dest	False	False	False	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	False	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	False	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	Lift 1 Sched.Max
New Sched 1.Min	4	4	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Stop	Stop	Stop	Stop
L1 Status.Dir	Down	Up	Up	Down
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 64. Lift Activity Behavior - A231 Part 8

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop				True
Lift 1 Sched.F12 Dest				True
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				False
Lift 1 Sched.F12 Up				False
Lift 1 Sched.F12 Down				
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	> Lift Status.L1 Loc		= Lift Status.L1 Loc	= Lift Status.L1 Loc
Lift 1 Sched.Min		< Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift 1 Sched.Max
Motor Ind.L1 Ind	Idle	Idle		
Lift Status.L1 Dir				Up
Lift Status.L1 Loc			1	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	False	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	False	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest	Lift 1 Sched.F12 Dest	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	False	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	0	0
New Sched 1.Min	Lift 1 Sched.Min	Lift 1 Sched.Min	4	4
L1 Status.Motor	Run	Run	Stop	Stop
L1 Status.Dir	Up	Down	Up	Down
L1 Status.Loc	Lift Status.L1 Loc + 1	Lift Status.L1 Loc - 1	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 65. Lift Activity Behavior - A231 Part 9

Process Lift 1				
Emer Signal.L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	True		True	True
Lift 1 Sched.F12 Dest	True			
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up	False		True	True
Lift 1 Sched.F12 Down	False		False	True
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	= Lift 1 Sched.Min	= Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift Status.L1 Loc
Lift 1 Sched.Min	= Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift 1 Sched.Max
Motor Ind.L1 Ind				
Lift Status.L1 Dir	Down		Up	Up
Lift Status.L1 Loc	2	3	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	False	Lift 1 Sched.F12 Stop	False	Lift 1 Sched.F12 Stop
New Sched 1.F12 Dest	False	Lift 1 Sched.F12 Dest	False	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	False	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	False	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	False	False
New Sched 1.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	False	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	0	0	0	Lift 1 Sched.Max
New Sched 1.Min	4	4	4	Lift 1 Sched.Min
L1 Status.Motor	Stop	Stop	Stop	Stop
L1 Status.Dir	Up	Down	Up	Up
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 66. Lift Activity Behavior - A231 Part 10

Process Lift 1				
Emer Signal L1 Value	False	False	False	False
Lift 1 Sched.F11 Stop				
Lift 1 Sched.F11 Dest				
Lift 1 Sched.F12 Stop	True	True	True	True
Lift 1 Sched.F12 Dest				
Lift 1 Sched.F13 Stop				
Lift 1 Sched.F13 Dest				
Lift 1 Sched.F11 Up				
Lift 1 Sched.F12 Up	False	False	True	True
Lift 1 Sched.F12 Down	True	True	True	False
Lift 1 Sched.F13 Down				
Lift 1 Sched.Max	= Lift Status.L1 Loc	= Lift 1 Sched.Min	= Lift 1 Sched.Min	= Lift 1 Sched.Min
Lift 1 Sched.Min	= Lift 1 Sched.Max	= Lift Status.L1 Loc	= Lift Status.L1 Loc	= Lift Status.L1 Loc
Motor Ind.L1 Ind				
Lift Status.L1 Dir	Up	Down	Down	Down
Lift Status.L1 Loc	2	2	2	2
New Sched 1.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop	Lift 1 Sched.F11 Stop
New Sched 1.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest	Lift 1 Sched.F11 Dest
New Sched 1.F12 Stop	False	False	Lift 1 Sched.F12 Stop	False
New Sched 1.F12 Dest	False	False	False	False
New Sched 1.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop	Lift 1 Sched.F13 Stop
New Sched 1.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest	Lift 1 Sched.F13 Dest
New Sched 1.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up	Lift 1 Sched.F11 Up
New Sched 1.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	Lift 1 Sched.F12 Up	False
New Sched 1.F12 Down	False	False	False	Lift 1 Sched.F12 Down
New Sched 1.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down	Lift 1 Sched.F13 Down
New Sched 1.Max	0	0	Lift 1 Sched.Max	0
New Sched 1.Min	4	4	Lift 1 Sched.Min	4
L1 Status Motor	Stop	Stop	Stop	Stop
L1 Status Dir	Down	Down	Down	Up
L1 Status Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 67. Lift Activity Behavior - A231 Part 11

Process Lift 1				
Emer Signal L1 Value	False	False	False	False
Lift 1 Sched.FI1 Stop				
Lift 1 Sched.FI1 Dest				
Lift 1 Sched.FI2 Stop				
Lift 1 Sched.FI2 Dest				
Lift 1 Sched.FI3 Stop				
Lift 1 Sched.FI3 Dest				
Lift 1 Sched.FI1 Up				True
Lift 1 Sched.FI2 Up			True	False
Lift 1 Sched.FI2 Down	True	False		
Lift 1 Sched.FI3 Down		True		
Lift 1 Sched.Max				= Lift 1 Sched.Min
Lift 1 Sched.Min	≠ Lift 1 Sched.Max	≠ Lift 1 Sched.Max	≠ Lift 1 Sched.Max	≠ Lift 1 Sched.Max
Motor Ind.L1 Ind	Run	Run	Run	Run
Lift Status.L1 Dir	Down	Down	Up	Up
Lift Status.L1 Loc	1	1	3	3
New Sched 1.FI1 Stop	False	False	Lift 1 Sched.FI1 Stop	Lift 1 Sched.FI1 Stop
New Sched 1.FI1 Dest	False	False	Lift 1 Sched.FI1 Dest	Lift 1 Sched.FI1 Dest
New Sched 1.FI2 Stop	Lift 1 Sched.FI2 Stop	Lift 1 Sched.FI2 Stop	Lift 1 Sched.FI2 Stop	Lift 1 Sched.FI2 Stop
New Sched 1.FI2 Dest	Lift 1 Sched.FI2 Dest	Lift 1 Sched.FI2 Dest	Lift 1 Sched.FI2 Dest	Lift 1 Sched.FI2 Dest
New Sched 1.FI3 Stop	Lift 1 Sched.FI3 Stop	Lift 1 Sched.FI3 Stop	False	False
New Sched 1.FI3 Dest	Lift 1 Sched.FI3 Dest	Lift 1 Sched.FI3 Dest	False	False
New Sched 1.FI1 Up	False	False	Lift 1 Sched.FI1 Up	Lift 1 Sched.FI1 Up
New Sched 1.FI2 Up	Lift 1 Sched.FI2 Up	Lift 1 Sched.FI2 Up	Lift 1 Sched.FI2 Up	Lift 1 Sched.FI2 Up
New Sched 1.FI2 Down	Lift 1 Sched.FI2 Down	Lift 1 Sched.FI2 Down	Lift 1 Sched.FI2 Down	Lift 1 Sched.FI2 Down
New Sched 1.FI3 Down	Lift 1 Sched.FI3 Down	Lift 1 Sched.FI3 Down	False	False
New Sched 1.Max	Lift 1 Sched.Max	Lift 1 Sched.Max	2	1
New Sched 1.Min	2	3	Lift 1 Sched.Min	Lift 1 Sched.Min
L1 Status.Motor	Stop	Stop	Stop	Stop
L1 Status.Dir	Up	Up	Down	Down
L1 Status.Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc	Lift Status.L1 Loc

Table 68. Lift Activity Behavior - A232 Part 1

Process Lift 2				
Emer Signal L2 Value	True	False	False	False
Lift 2 Sched F11 Stop				
Lift 2 Sched F11 Dest				
Lift 2 Sched F12 Stop				
Lift 2 Sched F12 Dest				
Lift 2 Sched F13 Stop				
Lift 2 Sched F13 Dest				
Lift 2 Sched F11 Up				
Lift 2 Sched F12 Up				
Lift 2 Sched F12 Down		False		
Lift 2 Sched F13 Down		False		
Lift 2 Sched Max			0	> Lift Status L2 Loc
Lift 2 Sched Min		≠ Lift 2 Sched Max	4	
Motor Ind L2 Ind		Run		Stop
Lift Status L2 Dir		Down		Up
Lift Status L2 Loc		1		1
New Sched 2 F11 Stop	Lift 2 Sched F11 Stop	False	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop
New Sched 2 F11 Dest	Lift 2 Sched F11 Dest	False	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest
New Sched 2 F12 Stop	Lift 2 Sched F12 Stop	Lift 2 Sched F12 Stop	Lift 2 Sched F12 Stop	Lift 2 Sched F12 Stop
New Sched 2 F12 Dest	Lift 2 Sched F12 Dest	Lift 2 Sched F12 Dest	Lift 2 Sched F12 Dest	Lift 2 Sched F12 Dest
New Sched 2 F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop
New Sched 2 F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest
New Sched 2 F11 Up	Lift 2 Sched F11 Up	False	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up
New Sched 2 F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up
New Sched 2 F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down
New Sched 2 F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down
New Sched 2 Max	Lift 2 Sched Max	Lift 2 Sched Max	Lift 2 Sched Max	Lift 2 Sched Max
New Sched 2 Min	Lift 2 Sched Min	4	Lift 2 Sched Min	Lift 2 Sched Min
L2 Status Motor	Stop	Stop	Idle	Run
L2 Status Dir	Lift Status L2 Dir	Up	Lift Status L2 Dir	Up
L2 Status Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc + 1

Table 69. Lift Activity Behavior - A232 Part 2

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched F11 Stop				
Lift 2 Sched F11 Dest				
Lift 2 Sched F12 Stop	False	True		True
Lift 2 Sched F12 Dest		True		True
Lift 2 Sched F13 Stop				
Lift 2 Sched F13 Dest				
Lift 2 Sched F11 Up				
Lift 2 Sched F12 Up		False		False
Lift 2 Sched F12 Down		False		False
Lift 2 Sched F13 Down				
Lift 2 Sched Max		> Lift Status L2 Loc	> Lift Status L2 Loc	= Lift Status L2 Loc
Lift 2 Sched Min				≠ Lift 2 Sched Max
Motor Ind L2 Ind	Run	Run	Stop	
Lift Status L2 Dir	Up	Up	Up	Up
Lift Status L2 Loc	2	2	2	2
New Sched 2 F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop
New Sched 2 F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest
New Sched 2 F12 Stop	Lift 2 Sched F12 Stop	False	Lift 2 Sched F12 Stop	False
New Sched 2 F12 Dest	Lift 2 Sched F12 Dest	False	Lift 2 Sched F12 Dest	False
New Sched 2 F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop
New Sched 2 F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest
New Sched 2 F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up
New Sched 2 F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up
New Sched 2 F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down
New Sched 2 F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down
New Sched 2 Max	Lift 2 Sched Max	Lift 2 Sched Max	Lift 2 Sched Max	0
New Sched 2 Min	Lift 2 Sched Min	Lift 2 Sched Min	Lift 2 Sched Min	Lift 2 Sched Min
L2 Status Motor	Run	Stop	Run	Stop
L2 Status Dir	Up	Up	Up	Down
L2 Status Loc	Lift Status L2 Loc + 1	Lift Status L2 Loc	Lift Status L2 Loc + 1	Lift Status L2 Loc

Table 70. Lift Activity Behavior - A232 Part 3

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched F11 Stop				
Lift 2 Sched F11 Dest				
Lift 2 Sched F12 Stop		True	False	True
Lift 2 Sched F12 Dest		True		True
Lift 2 Sched F13 Stop				
Lift 2 Sched F13 Dest				
Lift 2 Sched F11 Up				
Lift 2 Sched F12 Up		False		False
Lift 2 Sched F12 Down		False		False
Lift 2 Sched F13 Down				
Lift 2 Sched Max				≠ Lift 2 Sched Min
Lift 2 Sched Min	< Lift Status L2 Loc	< Lift Status L2 Loc		= Lift Status L2 Loc
Motor Ind L2 Ind	Stop	Run	Run	
Lift Status L2 Dir	Down	Down	Down	Down
Lift Status L2 Loc	2	2	2	2
New Sched 2 F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop
New Sched 2 F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest
New Sched 2 F12 Stop	Lift 2 Sched F12 Stop	False	Lift 2 Sched F12 Stop	False
New Sched 2 F12 Dest	Lift 2 Sched F12 Dest	False	Lift 2 Sched F12 Dest	False
New Sched 2 F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop
New Sched 2 F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest
New Sched 2 F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up
New Sched 2 F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up
New Sched 2 F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down
New Sched 2 F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down
New Sched 2 Max	Lift 2 Sched Max	Lift 2 Sched Max	Lift 2 Sched Max	Lift 2 Sched Max
New Sched 2 Min	Lift 2 Sched Min	Lift 2 Sched Min	Lift 2 Sched Min	4
L2 Status Motor	Run	Stop	Run	Stop
L2 Status Dir	Down	Down	Down	Up
L2 Status Loc	Lift Status L2 Loc - 1	Lift Status L2 Loc	Lift Status L2 Loc - 1	Lift Status L2 Loc

Table 71. Lift Activity Behavior - A232 Part 4

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop			True	True
Lift 2 Sched.F12 Dest				
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up	False			
Lift 2 Sched.F12 Up	False		True	True
Lift 2 Sched.F12 Down			False	True
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max	≠ Lift 2 Sched.Min		> Lift Status.L2 Loc	> Lift Status.L2 Loc
Lift 2 Sched.Min		< Lift Status.L2 Loc		
Motor Ind.L2 Ind	Run	Stop	Run	Run
Lift Status.L2 Dir	Up	Down	Up	Up
Lift Status.L2 Loc	3	3	2	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	False	Lift 2 Sched.F12 Stop
New Sched 2.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest	False	False
New Sched 2.F13 Stop	False	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	False	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	False	False
New Sched 2.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down
New Sched 2.F13 Down	False	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	0	Lift 2 Sched.Max	Lift 2 Sched.Max	Lift 2 Sched.Max
New Sched 2.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	Lift 2 Sched.Min
L2 Status.Motor	Stop	Run	Stop	Stop
L2 Status.Dir	Down	Down	Up	Up
L2 Status.Loc	Lift Status.L2 Loc	Lift Status.L2 Loc - 1	Lift Status.L2 Loc	Lift Status.L2 Loc

Table 72. Lift Activity Behavior - A232 Part 5

Process Lift 2				
Emer Signal.L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop	True	True	True	True
Lift 2 Sched.F12 Dest	False			
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				
Lift 2 Sched.F12 Up	False	True	True	False
Lift 2 Sched.F12 Down	True	False	True	True
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max	> Lift Status.L2 Loc	= Lift Status.L2 Loc	= Lift Status.L2 Loc	= Lift Status.L2 Loc
Lift 2 Sched.Min		≠ Lift 2 Sched.Max	≠ Lift 2 Sched.Max	≠ Lift 2 Sched.Max
Motor Ind.L2 Ind	Run			
Lift Status.L2 Dir	Up	Up	Up	Up
Lift Status.L2 Loc	2	2	2	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	Lift 2 Sched.F12 Stop	False	Lift 2 Sched.F12 Stop	False
New Sched 2.F12 Dest	Lift 2 Sched.F12 Dest	False	False	False
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	False	False	Lift 2 Sched.F12 Up
New Sched 2.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	False
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	Lift 2 Sched.Max	0	0	0
New Sched 2.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	Lift 2 Sched.Min
L2 Status.Motor	Motor Ind.L2 Ind	Stop	Stop	Stop
L2 Status.Dir	Lift Status.L2 Dir	Up	Up	Down
L2 Status.Loc	Lift Status.L2 Loc + 1	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc

Table 73. Lift Activity Behavior - A232 Part 6

Process Lift 2				
Emer Signal.L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop	True	True	True	True
Lift 2 Sched.F12 Dest			False	
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				
Lift 2 Sched.F12 Up	False	True	True	False
Lift 2 Sched.F12 Down	True	True	False	True
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max				≠ Lift 2 Sched.Min
Lift 2 Sched.Min	< Lift Status.L2 Loc	< Lift Status.L2 Loc	< Lift Status.L2 Loc	= Lift Status.L2 Loc
Motor Ind.L2 Ind	Run	Run	Run	
Lift Status.L2 Dir	Down	Down	Down	Down
Lift Status.L2 Loc	2	2	2	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	False	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	False
New Sched 2.F12 Dest	False	False	Lift 2 Sched.F12 Dest	False
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up
New Sched 2.F12 Down	False	False	Lift 2 Sched.F12 Down	False
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	Lift 2 Sched.Max
New Sched 2.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	4
L2 Status.Motor	Stop	Stop	Motor Ind.L2 Ind	Stop
L2 Status.Dir	Down	Down	Lift Status.L2 Dir	Down
L2 Status.Loc	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc - 1	Lift Status.L2 Loc

Table 74. Lift Activity Behavior - A232 Part 7

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop	True	True	True	True
Lift 2 Sched.F12 Dest			True	True
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				
Lift 2 Sched.F12 Up	True	True	False	True
Lift 2 Sched.F12 Down	True	False	True	False
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max	≠ Lift 2 Sched.Min	≠ Lift 2 Sched.Min	> Lift Status.L2 Loc	
Lift 2 Sched.Min	= Lift Status.L2 Loc	= Lift Status.L2 Loc		< Lift Status.L2 Loc
Motor Ind.L2 Ind			Run	Run
Lift Status.L2 Dir	Down	Down	Up	Down
Lift Status.L2 Loc	2	2	2	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	Lift 2 Sched.F12 Stop	False	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop
New Sched 2.F12 Dest	False	False	False	False
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	False	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up
New Sched 2.F12 Down	False	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	Lift 2 Sched.Max
New Sched 2.Min	4	4	Lift 2 Sched.Min	Lift 2 Sched.Min
L2 Status.Motor	Stop	Stop	Stop	Stop
L2 Status.Dir	Down	Up	Up	Down
L2 Status.Loc	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc

Table 75. Lift Activity Behavior - A232 Part 8

Process Lift 2				
Emer Signal.L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop				True
Lift 2 Sched.F12 Dest				True
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				
Lift 2 Sched.F12 Up				False
Lift 2 Sched.F12 Down				False
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max	> Lift Status.L2 Loc		= Lift Status.L2 Loc	= Lift Status.L2 Loc
Lift 2 Sched.Min		< Lift Status.L2 Loc	= Lift Status.L2 Loc	= Lift 2 Sched.Max
Motor Ind.L2 Ind	Idle	Idle		
Lift Status.L2 Dir				Up
Lift Status.L2 Loc			1	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	False	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	False	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	False
New Sched 2.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest	False
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	False	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up
New Sched 2.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	0	0
New Sched 2.Min	Lift 2 Sched.Min	Lift 2 Sched.Min	4	4
L2 Status.Motor	Run	Run	Stop	Stop
L2 Status.Dir	Up	Down	Up	Down
L2 Status.Loc	Lift Status.L2 Loc + 1	Lift Status.L2 Loc - 1	Lift Status.L2 Loc	Lift Status.L2 Loc

Table 76. Lift Activity Behavior - A232 Part 9

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched F11 Stop				
Lift 2 Sched F11 Dest				
Lift 2 Sched F12 Stop	True		True	True
Lift 2 Sched F12 Dest	True			
Lift 2 Sched F13 Stop				
Lift 2 Sched F13 Dest				
Lift 2 Sched F11 Up				
Lift 2 Sched F12 Up	False		True	True
Lift 2 Sched F12 Down	False		False	True
Lift 2 Sched F13 Down				
Lift 2 Sched Max	= Lift 2 Sched Min	= Lift Status L2 Loc	= Lift Status L2 Loc	= Lift Status L2 Loc
Lift 2 Sched Min	= Lift Status L2 Loc	= Lift Status L2 Loc	= Lift Status L2 Loc	= Lift 2 Sched Max
Motor Ind. L2 Ind				
Lift Status L2 Dir	Down		Up	Up
Lift Status L2 Loc	2	3	2	2
New Sched 2 F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop	Lift 2 Sched F11 Stop
New Sched 2 F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest	Lift 2 Sched F11 Dest
New Sched 2 F12 Stop	False	Lift 2 Sched F12 Stop	False	Lift 2 Sched F12 Stop
New Sched 2 F12 Dest	False	Lift 2 Sched F12 Dest	False	False
New Sched 2 F13 Stop	Lift 2 Sched F13 Stop	False	Lift 2 Sched F13 Stop	Lift 2 Sched F13 Stop
New Sched 2 F13 Dest	Lift 2 Sched F13 Dest	False	Lift 2 Sched F13 Dest	Lift 2 Sched F13 Dest
New Sched 2 F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up	Lift 2 Sched F11 Up
New Sched 2 F12 Up	Lift 2 Sched F12 Up	Lift 2 Sched F12 Up	False	False
New Sched 2 F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down	Lift 2 Sched F12 Down
New Sched 2 F13 Down	Lift 2 Sched F13 Down	False	Lift 2 Sched F13 Down	Lift 2 Sched F13 Down
New Sched 2 Max	0	0	0	Lift 2 Sched Max
New Sched 2 Min	4	4	4	Lift 2 Sched Min
L2 Status Motor	Stop	Stop	Stop	Stop
L2 Status Dir	Up	Down	Up	Up
L2 Status Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc

Table 77. Lift Activity Behavior - A232 Part 10

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop	True	True	True	True
Lift 2 Sched.F12 Dest				
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				
Lift 2 Sched.F12 Up	False	False	True	True
Lift 2 Sched.F12 Down	True	True	True	False
Lift 2 Sched.F13 Down				
Lift 2 Sched.Max	= Lift Status.L2 Loc	= Lift 2 Sched.Min	= Lift 2 Sched.Min	= Lift 2 Sched.Min
Lift 2 Sched.Min	= Lift 2 Sched.Max	= Lift Status.L2 Loc	= Lift Status.L2 Loc	= Lift Status.L2 Loc
Motor Ind.L2 Ind				
Lift Status.L2 Dir	Up	Down	Down	Down
Lift Status.L2 Loc	2	2	2	2
New Sched 2.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	False	False	Lift 2 Sched.F12 Stop	False
New Sched 2.F12 Dest	False	False	False	False
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest
New Sched 2.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 1 Sched.F12 Up	False
New Sched 2.F12 Down	False	False	False	Lift 2 Sched.F12 Down
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down
New Sched 2.Max	0	0	Lift 2 Sched.Max	0
New Sched 2.Min	4	4	Lift 2 Sched.Min	4
L2 Status.Motor	Stop	Stop	Stop	Stop
L2 Status.Dir	Down	Down	Down	Up
L2 Status.Loc	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc	Lift Status.L2 Loc

Table 78. Lift Activity Behavior - A232 Part 11

Process Lift 2				
Emer Signal L2 Value	False	False	False	False
Lift 2 Sched.F11 Stop				
Lift 2 Sched.F11 Dest				
Lift 2 Sched.F12 Stop				
Lift 2 Sched.F12 Dest				
Lift 2 Sched.F13 Stop				
Lift 2 Sched.F13 Dest				
Lift 2 Sched.F11 Up				True
Lift 2 Sched.F12 Up			True	False
Lift 2 Sched.F12 Down	True	False		
Lift 2 Sched.F13 Down		True		
Lift 2 Sched.Max				= Lift 2 Sched.Min
Lift 2 Sched.Min	≠ Lift 2 Sched.Max	≠ Lift 2 Sched.Max	≠ Lift 2 Sched.Max	≠ Lift 2 Sched.Max
Motor Ind L2 Ind	Run	Run	Run	Run
Lift Status L2 Dir	Down	Down	Up	Up
Lift Status L2 Loc	1	1	3	3
New Sched 2.F11 Stop	False	False	Lift 2 Sched.F11 Stop	Lift 2 Sched.F11 Stop
New Sched 2.F11 Dest	False	False	Lift 2 Sched.F11 Dest	Lift 2 Sched.F11 Dest
New Sched 2.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop	Lift 2 Sched.F12 Stop
New Sched 2.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest	Lift 2 Sched.F12 Dest
New Sched 2.F13 Stop	Lift 2 Sched.F13 Stop	Lift 2 Sched.F13 Stop	False	False
New Sched 2.F13 Dest	Lift 2 Sched.F13 Dest	Lift 2 Sched.F13 Dest	False	False
New Sched 2.F11 Up	False	False	Lift 2 Sched.F11 Up	Lift 2 Sched.F11 Up
New Sched 2.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up	Lift 2 Sched.F12 Up
New Sched 2.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down	Lift 2 Sched.F12 Down
New Sched 2.F13 Down	Lift 2 Sched.F13 Down	Lift 2 Sched.F13 Down	False	False
New Sched 2.Max	Lift 2 Sched.Max	Lift 2 Sched.Max	2	1
New Sched 2.Min	2	3	Lift 2 Sched.Min	Lift 2 Sched.Min
L2 Status.Motor	Stop	Stop	Stop	Stop
L2 Status.Dir	Up	Up	Down	Down
L2 Status.Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc	Lift Status L2 Loc

Table 79. Lift Activity Behavior - A233

Set System Status	
L1 Status.Motor	
L1 Status.Dir	
L1 Status.Loc	
L2 Status.Motor	
L2 Status.Dir	
L2 Status.Loc	
Motor Ind.L1 Ind	L1 Status.Motor
Motor Ind.L2 Ind	L2 Status.Motor
Lift Status.L1 Loc	L1 Status.Loc
Lift Status.L1 Dir	L1 Status.Dir
Lift Status.L2 Loc	L2 Status.Loc
Lift Status.L2 Dir	L2 Status.Dir

Appendix C. The Heating System Source Code

C.1 Type Definitions Package

```
1  -----
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       26 Jul 91
5  --* VERSION:    1.2
6  --* FILENAME:   TYPES.VHD
7  --*
8  --* FUNCTION:   Package for encapsulating the type definitions used in
9  --*              the Heating System Problem
10 --*
11 --* AUTHOR:     Capt Dan Eickmeier
12 --* HISTORY:    V1.0 - 22 Jul 91 - Initial version
13 --*              V1.1 - 26 Jul 91 - Made it into a Package
14 --*              V1.2 - 26 Jul 91 - Removed reserved word use of
15 --*              "ON" and "OPEN" as variables.
16 --*              26 Jul 91 - Compiles Successfully
17 --*
18  -----
19 package Heater_Package is
20
21     -- BASIC TYPES
22
23     type Combustion_Sensor_Type is (SAFE, UNSAFE);
24     type Fuel_Flow_Type is (SAFE, UNSAFE);
25     type Master_Switch_Type is (HEAT, OFF);
26     type Motor_Speed_Type is (BELOW_THRESHOLD, THRESHOLD);
27     type Water_Temp_Type is (BELOW_THRESHOLD, THRESHOLD);
28     type Motor_Control_Type is (OFF_STATE, ON_STATE);
29     type On_Off_Indicator_Type is (OFF_STATE, ON_STATE);
30     type Ignition_Type is (OFF_STATE, ON_STATE);
31     type Oil_Valve_Control_Type is (CLOSED_STATE, OPEN_STATE);
32     type Water_Valve_Control_Type is (CLOSED_STATE, OPEN_STATE);
33
34 end Heater_Package;
```

C.2 Architecture Structure of System

```

1  -----
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       10 Aug 91
5  --* VERSION:    1.1
6  --* FILENAME:   HEATER_SYSTEM.VHD
7  --*
8  --* FUNCTION:   Entity and Structural description for the System Entity
9  --*              which connects the Control_Heater (A-0) entity to the
10 --*              Timers and Testbench entities.
11 --*
12 --* AUTHOR:     Capt Dan Eickmeier
13 --* HISTORY:    V1.0 - 10 Aug 91 - Initial Version
14 --*              - 10 Aug 91 - Compiles Successfully
15 --*              V1.1 - 10 Aug 91 - Added initial conditions on signals
16 --*              - 10 Aug 91 - Compiles Successfully
17 --*
18 -----

19 use Work.Heater_Package.all;

20 entity system is
21 end system;

22 architecture structure of system is

23     -- COMPONENT DECLARATIONS

24     component Control_Heater_UUT
25     port (Air_Temp      : in Integer;
26           Combustion_Sensor : in Combustion_Sensor_Type;
27           Fuel_Flow      : in Fuel_Flow_Type;
28           Desired_Air_Temp : in Integer;
29           Master_Switch   : in Master_Switch_Type;
30           Five_Sec_Timer  : in Boolean;
31           Five_Min_Timer  : in Boolean;
32           Motor_Speed     : in Motor_Speed_Type;
33           Water_Temp      : in Water_Temp_Type;
34           --
35           Motor_Control   : out Motor_Control_Type := OFF_STATE;
36           On_Off_Indicator : out On_Off_Indicator_Type := OFF_STATE;
37           Ignition         : out Ignition_Type := OFF_STATE;
38           Oil_Valve_Control : out Oil_Valve_Control_Type := CLOSED_STATE;
39           Water_Valve_Control : out Water_Valve_Control_Type := CLOSED_STATE);
40     end component;

41     component Delay_Five_Min_UUT
42     generic ( Five_Min_Delay : Time );
43     port ( Motor_Control : in Motor_Control_Type;

```

```

44         Five_Min_Timer : out Boolean := TRUE);
45     end component;

46     component Delay_Five_Sec_UUT
47         generic ( Five_Sec_Delay : Time );

48         port ( Oil_Valve_Control : in Oil_Valve_Control_Type;
49               Five_Sec_Timer     : out Boolean := TRUE);
50     end component;

51     component testbench
52         port (Air_Temp           : out Integer := 0;
53               Combustion_Sensor : out Combustion_Sensor_Type := SAFE;
54               Fuel_Flow         : out Fuel_Flow_Type := SAFE;
55               Desired_Air_Temp  : out Integer := 0;
56               Master_Switch     : out Master_Switch_Type := HEAT;
57               Motor_Speed       : out Motor_Speed_Type := BELOW_THRESHOLD;
58               Water_Temp        : out Water_Temp_Type := BELOW_THRESHOLD;
59               --
60               Motor_Control     : in Motor_Control_Type;
61               On_Off_Indicator  : in On_Off_Indicator_Type;
62               Ignition          : in Ignition_Type;
63               Oil_Valve_Control : in Oil_Valve_Control_Type;
64               Water_Valve_Control : in Water_Valve_Control_Type);

65     end component;

66     -- SIGNAL DECLARATIONS

67     signal I1      : Integer;
68     signal C1      : Combustion_Sensor_Type;
69     signal C2      : Fuel_Flow_Type;
70     signal C3      : Integer;
71     signal C4      : Master_Switch_Type;
72     signal C5      : Boolean;
73     signal C6      : Boolean;
74     signal C7      : Motor_Speed_Type;
75     signal C8      : Water_Temp_Type;
76     signal O1      : Motor_Control_Type;
77     signal O2      : On_Off_Indicator_Type;
78     signal O3      : Ignition_Type;
79     signal O4      : Oil_Valve_Control_Type;
80     signal O5      : Water_Valve_Control_Type;

81     begin

82         -- COMPONENT INSTANTIATIONS

83         Device: Control_Heater_UUT port map (
84             I1,
85             C1, C2, C3, C4, C5, C6, C7, C8,

```

```

86             01, 02, 03, 04, 05);

87     Timer1: Delay_Five_Min_UUT
88             generic map (5 min)
89             port map (
90                 01,
91                 C6);

92     Timer2: Delay_Five_Sec_UUT
93             generic map (5 sec)
94             port map (
95                 04,
96                 C5);

97     Bench: Testbench port map (
98         I1,
99         C1, C2, C3, C4, C7, C8,
100        01, 02, 03, 04, 05);

101 end;

```

C.3 Entity Declaration for Control_Heater (A-0)

```

1  --*****
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       29 Jul 91
5  --* VERSION:    1.3
6  --* FILENAME:   HEAT_A-0.VHD
7  --*
8  --* FUNCTION:   Entity declaration for the A-0 Level SADT Diagram
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 22 Jul 91 - Initial version
12 --*              V1.1 - 26 Jul 91 - Removed Reserved Word Usage
13 --*              V1.2 - 27 Jul 91 - INOUT declarations added for signals
14 --*              - 27 Jul 91 - Compiles Successfully
15 --*              V1.3 - 29 Jul 91 - Removed INOUT - Made Local Signals
16 --*              - 29 Jul 91 - Compiles Successfully
17 --*
18 --*****

19 use Work.Heater_Package.all;

20 entity Control_Heater is
21     port (Air_Temp      : in Integer;
22           Combustion_Sensor : in Combustion_Sensor_Type;
23           Fuel_Flow      : in Fuel_Flow_Type;
24           Desired_Air_Temp : in Integer;
25           Master_Switch  : in Master_Switch_Type;
26           Five_Sec_Timer : in Boolean;
27           Five_Min_Timer : in Boolean;
28           Motor_Speed    : in Motor_Speed_Type;
29           Water_Temp     : in Water_Temp_Type;
30           --
31           Motor_Control  : out Motor_Control_Type;
32           On_Off_Indicator : out On_Off_Indicator_Type;
33           Ignition       : out Ignition_Type;
34           Oil_Valve_Control : out Oil_Valve_Control_Type;
35           Water_Valve_Control : out Water_Valve_Control_Type);
36 end Control_Heater;

```

C.4 Architecture Structure of Control_Heater (A-0)

```

1  --*****
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       29 Jul 91
5  --* VERSION:    1.6
6  --* FILENAME:   HEAT_A-0_STRUCT.VHD
7  --*
8  --* FUNCTION:   Structural Description for Activities A1, A2, A3 and A4
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 22 Jul 91 - Initial version
12 --*              V1.1 - 26 Jul 91 - Added Component Instantiations
13 --*              V1.2 - 26 Jul 91 - Removed Reserved Word Usage
14 --*              V1.3 - 27 Jul 91 - Signal Declarations for Internal
15 --*                  interconnections
16 --*              V1.4 - 27 Jul 91 - INOUT declarations added for signals
17 --*                  - 27 Jul 91 - Compiles Successfully
18 --*              V1.5 - 28 Jul 91 - Shorten name of A4 Component due to
19 --*                  name being to long (VAX system limitation only)
20 --*                  - 28 Jul 91 - Compiles Successfully
21 --*              V1.6 - 29 Jul 91 - Removed INOUT - Made Local Signals
22 --*                  - 29 Jul 91 - Compiles Successfully
23  --*****

24 architecture structure of Control_Heater is
25
26      -- COMPONENT DECLARATIONS
27
28      component Shut_Off_Furnace
29      port ( Combustion_Sensor : in Combustion_Sensor_Type;
30              Fuel_Flow       : in Fuel_Flow_Type;
31              Shut_Down       : out Boolean);
32      end component;
33
34      component Control_Motor
35      port ( Air_Temp          : in Integer;
36              Shut_Down       : in Boolean;
37              Des_Air_Temp    : in Integer;
38              Master_Switch   : in Master_Switch_Type;
39              Five_Sec_Timer  : in Boolean;
40              Five_Min_Timer  : in Boolean;
41              Motor_Control   : out Motor_Control_Type;
42              Oil_Enable      : out Boolean );
43      end component;
44
45      component Control_Furnace_Ignition
46      port ( Oil_Enable       : in Boolean;
47              Motor_Control   : in Motor_Control_Type;
48              Shut_Down      : in Boolean;
49              Motor_Speed     : in Motor_Speed_Type;

```



```

46         On_Off_Indicator : out On_Off_Indicator_Type;
47         Ignition          : out Ignition_Type;
48         Oil_Valve_Control : out Oil_Valve_Control_Type);
49     end component;

50     component Control_Water_Valve -- A4
51     port ( Water_Temp          : in Water_Temp_Type;
52           Water_Valve_Control : out Water_Valve_Control_Type);
53     end component;

54     -- SIGNAL DECLARATIONS

55     signal A2_A3_Motor_Control : Motor_Control_Type;
56     signal A1_A2_A3_Shut_Down  : Boolean;
57     signal A2_A3_Oil_Enable    : Boolean;

58 begin

59     -- COMPONENT INSTANTIATIONS

60     A1: Shut_Off_Furnace port map (
61         Combustion_Sensor,
62         Fuel_Flow,
63         A1_A2_A3_Shut_Down );

64     A2: Control_Motor port map (
65         Air_Temp,
66         A1_A2_A3_Shut_Down,
67         Desired_Air_Temp,
68         Master_Switch,
69         Five_Sec_Timer,
70         Five_Min_Timer,
71         A2_A3_Motor_Control,
72         A2_A3_Oil_Enable );

73     A3: Control_Furnace_Ignition port map (
74         A2_A3_Oil_Enable,
75         A2_A3_Motor_Control,
76         A1_A2_A3_Shut_Down,
77         Motor_Speed,
78         On_Off_Indicator,
79         Ignition,
80         Oil_Valve_Control);

81     A4: Control_Water_Valve port map (
82         Water_Temp,
83         Water_Valve_Control);

84     -- SIGNAL ASSIGNMENT
85     Motor_Control <= A2_A3_Motor_Control;
86 end structure;

```

C.5 Entity Declarations for the A0 Level

```

1  -----
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       29 Jul 91
5  --* VERSION:    1.4
6  --* FILENAME:   HEAT_A0.VHD
7  --*
8  --* FUNCTION:   Entity declaration for the A0 Level SADT Diagram
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 22 Jul 91 - Initial Version
12 --*              V1.1 - 27 Jul 91 - Changed some declarations to INOUT
13 --*              V1.2 - 27 Jul 91 - Added Use Statements for each entity
14 --*              - 27 Jul 91 - Compiled Successfully
15 --*              V1.3 - 28 Jul 91 - Shorten name of A4 Component due to
16 --*              name being to long (VMS system limitation only)
17 --*              - 28 Jul 91 - Compiles Successfully
18 --*              V1.4 - 29 Jul 91 - Removed INOUT declarations
19 --*              - 29 Jul 91 - Compiles Successfully
20 --*
21 -----

22 use Work.Heater_Package.all;

23 entity Shut_Off_Furnace is                      -- A1
24     port ( Combustion_Sensor : in Combustion_Sensor_Type;
25           Fuel_Flow         : in Fuel_Flow_Type;
26           Shut_Down         : out Boolean);
27     end Shut_Off_Furnace;

28 use Work.Heater_Package.all;

29 entity Control_Motor is                          -- A2
30     port ( Air_Temp          : in Integer;
31           Shut_Down         : in Boolean;
32           Des_Air_Temp      : in Integer;
33           Master_Switch     : in Master_Switch_Type;
34           Five_Sec_Timer    : in Boolean;
35           Five_Min_Timer    : in Boolean;
36           Motor_Control     : out Motor_Control_Type;
37           Oil_Enable        : out Boolean );
38     end Control_Motor;

39 use Work.Heater_Package.all;

40 entity Control_Furnace_Ignition is                -- A3
41     port ( Oil_Enable        : in Boolean;
42           Motor_Control     : in Motor_Control_Type;
43           Shut_Down         : in Boolean,
44           Motor_Speed       : in Motor_Speed_Type;

```

```

45         On_Off_Indicator : out On_Off_Indicator_Type;
46         Ignition          : out Ignition_Type;
47         Oil_Valve_Control : out Oil_Valve_Control_Type);
48     end Control_Furnace_Ignition;

49 use Work.Heater_Package.all;

50 entity Control_Water_Valve is                                -- A4
51     port ( Water_Temp      : in Water_Temp_Type;
52           Water_Valve_Control : out Water_Valve_Control_Type);
53     end Control_Water_Valve;

```

C.6 Architecture Structures for the A0 Level

```

1  --*****
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       11 Aug 91
5  --* VERSION:    1.3
6  --* FILENAME:   HEAT_A0_STRUCT.VHD
7  --*
8  --* FUNCTION:   Structural Description for Activities A2 and A3
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 25 Jul 91 - Initial Version
12 --*              V1.1 - 27 Jul 91 - Changed some declarations to INOUT
13 --*              V1.2 - 28 Jul 91 - Added signal declaration for A3
14 --*              - 28 Jul 91 - Successfully Compiles
15 --*              V1.3 - 11 Aug 91 - Added Port Assignment for Oil_Enable
16 --*              - 11 Aug 91 - Successfully Compiles
17 --*
18 --*****

19 architecture Structure of Control_Motor is

20     -- COMPONENT DECLARATIONS

21     component Compare_Temperature
22     port ( Air_Temp      : in Integer;
23           Shut_Down     : in Boolean;
24           Des_Air_Temp  : in Integer;
25           Master_Switch : in Master_Switch_Type;
26           Oil_Enable    : out Boolean);
27     end component;

28     component Activate_Motor
29     port ( Oil_Enable    : in Boolean;
30           Five_Sec_Timer : in Boolean;
31           Five_Min_Timer : in Boolean;
32           Motor_Control  : out Motor_Control_Type );
33     end component;

34     --SIGNAL DECLARATIONS

35     signal A21_A22_Oil_Enable : Boolean;

36 begin

37     -- COMPONENT INSTANTIATIONS

38     A21: Compare_Temperature port map (
39         Air_Temp,
40         Shut_Down,
41         Des_Air_Temp,

```

```

42         Master_Switch,
43         A21_A22_Oil_Enable );

44     A22: Activate_Motor port map (                -- A22
45         A21_A22_Oil_Enable,
46         Five_Sec_Timer,
47         Five_Min_Timer,
48         Motor_Control );

49     -- SIGNAL ASSIGNMENT

50     Oil_Enable <= A21_A22_Oil_Enable;

51 end structure;

52 architecture structure of Control_Furnace_Ignition is

53     -- COMPONENT DECLARATIONS

54     component Monitor_Motor_Speed                -- A31
55     port ( Motor_Control      : in Motor_Control_Type;
56           Motor_Speed        : in Motor_Speed_Type;
57           On_Off_Indicator    : out On_Off_Indicator_Type;
58           Desired_Speed       : out Boolean);
59     end component;

60     component Activate_Ignition                -- A32
61     port ( Desired_Speed      : in Boolean;
62           Shut_Down           : in Boolean;
63           Ignition            : out Ignition_Type);
64     end component;

65     component Activate_Oil_Valve                -- A33
66     port ( Desired_Speed      : in Boolean;
67           Shut_Down           : in Boolean;
68           Oil_Enable          : in Boolean;
69           Oil_Valve_Control   : out Oil_Valve_Control_Type);
70     end component;

71     -- SIGNAL DECLARATIONS

72     signal Desired_Speed      : Boolean;

73 begin

74     -- COMPONENT INSTANTIATIONS

75     A31: Monitor_Motor_Speed port map                -- A31
76         ( Motor_Control,
77           Motor_Speed,
78           On_Off_Indicator,

```

```

79             Desired_Speed );

80     A32: Activate_Ignition port map                -- A32
81         ( Desired_Speed,
82           Shut_Down,
83           Ignition );

84     A33: Activate_Oil_Valve port map              -- A33
85         ( Desired_Speed,
86           Shut_Down,
87           Oil_Enable,
88           Oil_Valve_Control );

89 end structure;

```

C.7 Architecture Behavior for the A0 Level

```

1  -----
2  ---**
3  --- TITLE:      Heating System Problem      **
4  --- DATE:       29 Jul 91                    **
5  --- VERSION:    3.0                          **
6  --- FILENAME:   HEAT_A0_BEHAV.VHD           **
7  ---**
8  --- FUNCTION:   Behavior description for Activities A1 and A4  **
9  ---**
10 --- AUTHOR:     Capt Dan Eickmeier           **
11 --- HISTORY:    V1.0 - 22 Jul 91 - Initial Version      **
12 ---              V1.1 - 23 Jul 91 - Added Process Sensitivity Lists  **
13 ---              V2.0 - 25 Jul 91 - Changed Behavior from IF to CASE  **
14 ---              V2.1 - 26 Jul 91 - Eliminated the use of Reserved Words **
15 ---              V2.2 - 28 Jul 91 - Shorten name of A4 Component due to **
16 ---                  name being to long                    **
17 ---              28 Jul 91 - CASE Behavior Compiles Successfully  **
18 ---              V3.0 - 29 Jul 91 - Change back to IF Behavior      **
19 ---              29 Jul 91 - IF Behavior Compiles Successfully  **
20 ---**
21 -----

22 architecture Behavior of Shut_Off_Furnace is          --A1

23 begin
24     process (Combustion_Sensor, Fuel_Flow) begin

25         if Combustion_Sensor = UNSAFE then
26             Shut_Down <= TRUE;
27         elsif Fuel_Flow = UNSAFE then
28             Shut_Down <= TRUE;
29         elsif Combustion_Sensor = SAFE and Fuel_Flow = SAFE then
30             Shut_Down <= FALSE;
31         end if;
32     end process;
33 end Behavior;

34 architecture Behavior of Control_Water_Valve is      -- A4

35 begin
36     process (Water_Temp) begin

37         if Water_Temp = THRESHOLD then
38             Water_Valve_Control <= OPEN_STATE;
39         elsif Water_Temp = BELOW_THRESHOLD then
40             Water_Valve_Control <= CLOSED_STATE;
41         end if;
42     end process;
43 end Behavior;

```

C.8 Entity Declarations for the A2 Level

```

1  --*****
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       29 Jul 91
5  --* VERSION:    1.3
6  --* FILENAME:   HEAT_A2.VHD
7  --*
8  --* FUNCTION:   Entity declaration for the A2 Level SADT Diagram
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 22 Jul 91 - Initial Version
12 --*              V1.1 - 27 Jul 91 - Changed some declarations to INOUT
13 --*              V1.2 - 27 Jul 91 - Added Use Statements for each entity
14 --*              - 27 Jul 91 - Compiled Successfully
15 --*              V1.3 - 29 Jul 91 - Removed INOUT declarations
16 --*              - 29 Jul 91 - Compiled Successfully
17 --*
18 --*****

19 use Work.Heater_Package.all;

20 entity Compare_Temperature is                      -- A21
21     port ( Air_Temp      : in Integer;
22           Shut_Down     : in Boolean;
23           Des_Air_Temp   : in Integer;
24           Master_Switch : in Master_Switch_Type;
25           Oil_Enable     : out Boolean);
26     end Compare_Temperature;

27 use Work.Heater_Package.all;

28 entity Activate_Motor is                          -- A22
29     port ( Oil_Enable    : in Boolean;
30           Five_Sec_Timer : in Boolean;
31           Five_Min_Timer : in Boolean;
32           Motor_Control  : out Motor_Control_Type );
33     end Activate_Motor;

```


C.9 Architecture Behavior for the A2 Level

```

1  -----
2  ---
3  --- TITLE:      Heating System Problem
4  --- DATE:       13 Aug 91
5  --- VERSION:    3.2
6  --- FILENAME:   HEAT_A2_BEHAV.VHD
7  ---
8  --- FUNCTION:   Behavioral descriptions for Activities A21 and A22
9  ---
10 --- AUTHOR:     Capt Dan Eickmeier
11 --- HISTORY:    V1.0 - 22 Jul 91 - Initial Version
12 ---              V1.1 - 23 Jul 91 - Added Process Sensitivity Lists
13 ---              V2.0 - 25 Jul 91 - Changed Behavior from IF to CASE
14 ---              V2.1 - 26 Jul 91 - Eliminated the use of Reserved Words
15 ---              V2.2 - 23 Jul 91 - Changed Air Temp Behavior back to IF
16 ---                  because the > condition could not easily be
17 ---                  generated using a CASE construct
18 ---              - 28 Jul 91 - Compiled Successfully
19 ---              V3.0 - 29 Jul 91 - Changed back to IF Behavior
20 ---              - 29 Jul 91 - IF Behavior Compiled Successfully
21 ---              V3.1 - 12 Aug 91 - Changed Behavior from > to >=
22 ---              - 12 Aug 91 - Compiled Successfully
23 ---              V3.2 - 13 Aug 91 - Modified behavior of A22
24 ---              - 13 Aug 91 - Compiled Successfully
25 ---
26  -----

27  architecture Behavior of Compare_Temperature is                      -- A21

28  begin
29      process (Shut_Down, Master_Switch, Air_Temp, Des_Air_Temp) begin

30          if Shut_Down = TRUE then
31              Oil_Enable <= FALSE;
32          elsif Master_Switch = OFF then
33              Oil_Enable <= FALSE;
34          elsif Air_Temp >= (Des_Air_Temp + 2) then
35              Oil_Enable <= FALSE;
36          elsif Shut_Down = FALSE and Master_Switch = HEAT and
37              Des_Air_Temp > (Air_Temp + 2) then
38              Oil_Enable <= TRUE;
39          end if;

40      end process;
41  end Behavior;

42  architecture Behavior of Activate_Motor is                          -- A22

43  begin

```

```
44     process (Oil_Enable, Five_Min_Timer, Five_Sec_Timer) begin
45         if Oil_Enable = TRUE and Five_Min_Timer = TRUE then
46             Motor_Control <= ON_STATE;
47         elsif Oil_Enable = FALSE and Five_Sec_Timer = TRUE then
48             Motor_Control <= OFF_STATE;
49         end if;

50     end process;
51 end Behavior;
```

C.10 Entity Declarations for the A3 Level

```
1  --*****
2  --*
3  --* TITLE:      Heating System Problem
4  --* DATE:       29 Jul 91
5  --* VERSION:    1.3
6  --* FILENAME:   HEAT_A3.VHD
7  --*
8  --* FUNCTION:   Entity declaration for the A3 Level SADT Diagram
9  --*
10 --* AUTHOR:     Capt Dan Eickmeier
11 --* HISTORY:    V1.0 - 22 Jul 91 - Initial Version
12 --*              V1.1 - 27 Jul 91 - Changed some declarations to INOUT
13 --*              V1.2 - 27 Jul 91 - Added Use Statements for each entity
14 --*              - 27 Jul 91 - Compiled Successfully
15 --*              V1.3 - 29 Jul 91 - Removed INOUT declarations
16 --*              - 29 Jul 91 - Compiled Successfully
17 --*
18 --*****

19 use Work.Heater_Package.all;

20 entity Monitor_Motor_Speed is                      -- A31
21     port ( Motor_Control    : in Motor_Control_Type;
22            Motor_Speed      : in Motor_Speed_Type;
23            On_Off_Indicator : out On_Off_Indicator_Type;
24            Desired_Speed    : out Boolean);
25     end Monitor_Motor_Speed;

26 use Work.Heater_Package.all;

27 entity Activate_Ignition is                        -- A32
28     port ( Desired_Speed : in Boolean;
29            Shut_Down     : in Boolean;
30            Ignition      : out Ignition_Type);
31     end Activate_Ignition;

32 use Work.Heater_Package.all;

33 entity Activate_Oil_Valve is                      -- A33
34     port ( Desired_Speed    : in Boolean;
35            Shut_Down        : in Boolean;
36            Oil_Enable       : in Boolean;
37            Oil_Valve_Control : out Oil_Valve_Control_Type);
38     end Activate_Oil_Valve;
```

C.11 Architecture Behavior for the A3 Level

```

1  -----
2  ---*
3  ---* TITLE:      Heating System Problem
4  ---* DATE:       29 Jul 91
5  ---* VERSION:    3.0
6  ---* FILENAME:   HEAT_A3_BEHAV.VHD
7  ---*
8  ---* FUNCTION:   Behavioral descriptions for Activities A21 and A22
9  ---*
10 ---* AUTHOR:     Capt Dan Eickmeier
11 ---* HISTORY:    V1.0 - 22 Jul 91 - Initial Version
12 ---*              V1.1 - 23 Jul 91 - Added Process Sensitivity Lists
13 ---*              V2.0 - 25 Jul 91 - Changed Behavior from IF to CASE
14 ---*              V2.1 - 26 Jul 91 - Eliminated the use of Reserved Words
15 ---*              - 28 Jul 91 - CASE Behavior Compiled Successfully
16 ---*              V3.0 - 29 Jul 91 - Change back to IF Behavior
17 ---*              - 29 Jul 91 - IF Behavior Compiled Successfully
18 ---*
19 -----

20 architecture Behavior of Monitor_Motor_Speed is                      -- A31

21 begin
22     process (Motor_Control, Motor_Speed) begin

23         if Motor_Control = OFF_STATE then
24             On_Off_Indicator <= OFF_STATE;
25         elsif Motor_Control = ON_STATE and
26             Motor_Speed = BELOW_THRESHOLD then
27             On_Off_Indicator <= ON_STATE;
28             Desired_Speed <= False;
29         elsif Motor_Control = ON_STATE and
30             Motor_Speed = THRESHOLD then
31             On_Off_Indicator <= ON_STATE;
32             Desired_Speed <= TRUE;
33         end if;

34     end process;
35 end Behavior;

36 architecture Behavior of Activate_Ignition is                      -- A32

37 begin
38     process (Desired_Speed, Shut_Down) begin
39         if Shut_Down = TRUE then
40             Ignition <= OFF_STATE;
41         elsif Desired_Speed = TRUE and
42             Shut_Down = FALSE then
43             Ignition <= ON_STATE;

```

```

44         elsif Desired_Speed = FALSE and
45             Shut_Down = FALSE then
46             Ignition <= OFF_STATE;
47         end if;

48     end process;
49 end Behavior;

50 architecture Behavior of Activate_Oil_Valve is                                --A33

51 begin
52     process (Desired_Speed, Shut_Down, Oil_Enable) begin
53         if Shut_Down = TRUE then
54             Oil_Valve_Control <= CLOSED_STATE;
55         elsif Desired_Speed = FALSE then
56             Oil_Valve_Control <= CLOSED_STATE;
57         elsif Oil_Enable = FALSE then
58             Oil_Valve_Control <= CLOSED_STATE;
59         elsif Desired_Speed = TRUE and
60             Shut_Down = FALSE and
61             Oil_Enable = TRUE then
62             Oil_Valve_Control <= OPEN_STATE;
63         end if;

64     end process;
65 end Behavior;

```

C.12 Entity Declarations for Timers

```
1  --*****
2  --*
3  --*  TITLE:      Heating System Problem
4  --*  DATE:       09 Aug 91
5  --*  VERSION:    1.0
6  --*  FILENAME:   TIMERS.VHD
7  --*
8  --*  FUNCTION:   Entity description for the Timer Activities of the
9  --*               A-0 Diagram.
10 --*
11 --*  AUTHOR:     Capt Dan Eickmeier
12 --*  HISTORY:    V1.0 - 08 Aug 91 - Initial Version
13 --*               - 09 Aug 91 - Compiles Successfully
14 --*
15 --*****

16 use Work.Heater_Package.all;

17 entity Delay_Five_Min is
18     generic (Five_Min_Delay : Time := 5 min);

19     port    (Motor_Control : in Motor_Control_Type;
20             Five_Min_Timer : out Boolean := TRUE);
21 end Delay_Five_Min;

22 use Work.Heater_Package.all;

23 entity Delay_Five_Sec is
24     generic (Five_Sec_Delay : Time := 5 sec);

25     port    ( Oil_Valve_Control : in Oil_Valve_Control_Type;
26             Five_Sec_Timer      : out Boolean := TRUE);
27 end Delay_Five_Sec;
```

C.13 Architecture Behavior for Timers

```
1  -----
2  ---**
3  --- TITLE:      Heating System Problem      **
4  --- DATE:       09 Aug 91                    **
5  --- VERSION:    1.0                          **
6  --- FILENAME:   TIMERS_BEHAV.VHD            **
7  ---**
8  --- FUNCTION:   Behavior description for the Timer Activities of the **
9  ---              A-0 Diagram.                **
10 ---**
11 --- AUTHOR:     Capt Dan Eickmeier           **
12 --- HISTORY:    V1.0 - 08 Aug 91 - Initial Version **
13 ---              - 09 Aug 91 - Compiles Successfully **
14 ---**
15 -----
16 architecture Behavior of Delay_Five_Min is
17 begin
18     process (Motor_Control) begin
19
20         if (not Motor_Control'Stable) and
21             (Motor_Control = Off_State) then
22             Five_Min_Timer <= False after 0 ns,
23                 True after Five_Min_Delay;
24
25         elsif (not Motor_Control'Stable) and
26             (Motor_Control = On_State) then
27             Five_Min_Timer <= False;
28         end if;
29     end process;
30 end Behavior;
31
32 architecture Behavior of Delay_Five_Sec is
33 begin
34     process (Oil_Valve_Control) begin
35
36         if (not Oil_Valve_Control'Stable) and
37             (Oil_Valve_Control = Closed_State) then
38             Five_Sec_Timer <= False after 0 ns,
39                 True after Five_Sec_Delay;
40
41         elsif (not Oil_Valve_Control'Stable) and
42             (Oil_Valve_Control = Open_State) then
43             Five_Sec_Timer <= False;
44         end if;
45     end process;
46 end Behavior;
```

C.14 Entity and Architecture for the Testbench

```

--*****
--*                                                                 **
--* TITLE:      Heating System Problem                             **
--* DATE:       06 Oct 91                                           **
--* VERSION:    1.1                                                 **
--* FILENAME:   TESTBENCH.VHD                                       **
--*                                                                 **
--* FUNCTION:   Entity and Behavior declarations for the Heating System **
--*            Testbench.                                           **
--*                                                                 **
--* AUTHOR:     Capt Dan Eickmeier                                  **
--* HISTORY:    V1.0 - 09 Aug 91 - Initial Version                  **
--*            10 Aug 91 - Compiles Successfully                   **
--*            V1.1 - 06 Oct 91 - Added Text_IO routines to write text **
--*            comments to the Output                             **
--*            06 Oct 91 - Compiles Successfully                   **
--*                                                                 **
--*****

use Work.Heater_Package.all;

entity test is
    port (Air_Temp          : out Integer;
          Combustion_Sensor : out Combustion_Sensor_Type;
          Fuel_Flow         : out Fuel_Flow_Type;
          Desired_Air_Temp  : out Integer;
          Master_Switch     : out Master_Switch_Type;
          Motor_Speed       : out Motor_Speed_Type;
          Water_Temp        : out Water_Temp_Type;

          Motor_Control     : in Motor_Control_Type;
          On_Off_Indicator  : in On_Off_Indicator_Type;
          Ignition          : in Ignition_Type;
          Oil_Valve_Control : in Oil_Valve_Control_Type;
          Water_Valve_Control : in Water_Valve_Control_Type);

end test;

use Std.TextIO;

architecture bench of test is
begin
    stimulus: process

        subtype Message_Type is String (1 to 60);
        variable Message      : Message_Type := (others => ' ');
        variable NullString   : Message_Type := (others => ' ');
        variable StarString   : Message_Type := (others => '*');
        variable outline      : TextIO.Line;
        variable null_line    : TextIO.Line;

```



```

variable star_line    : TextIO.Line;

procedure Start_Message is
begin
    TextIO.Write(null_line, NullString);
    TextIO.Write(star_line, StarString);
    TextIO.WriteLine(TextIO.OUTPUT, null_line);
    TextIO.WriteLine(TextIO.OUTPUT, star_line);
end Start_Message;

procedure End_Message is
begin
    TextIO.Write(null_line, NullString);
    TextIO.Write(star_line, StarString);
    TextIO.WriteLine(TextIO.OUTPUT, star_line);
    TextIO.WriteLine(TextIO.OUTPUT, null_line);
end End_Message;

procedure Print_Message ( In_Message : Message_Type ) is
begin
    TextIO.Write(outline, In_Message);
    TextIO.WriteLine(TextIO.OUTPUT, outline);
end Print_Message;

begin
    -- TEST 1, Phase 1
    Start_Message;
    Message := "TEST CASE 1: Initial Startup conditions tested.  There      ";
    Print_Message (Message);
    Message := "are three phases in the test. Requirements R1, R2, and R3  ";
    Print_Message (Message);
    Message := "are tested.                                              ";
    Print_Message (Message);
    End_Message;

    Air_Temp      <= 75;
    Desired_Air_Temp <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow      <= Safe;
    Master_Switch   <= Heat;
    Motor_Speed     <= Below_Threshold;
    Water_Temp      <= Below_Threshold;

    WAIT for 1 sec;

    ASSERT
        (Motor_Control      = On_State      ) and
        (On_Off_Indicator    = On_State      ) and
        (Ignition            = Off_State     ) and
        (Oil_Valve_Control    = Closed_State) and

```

```

        (Water_Valve_Control = Closed_State)
REPORT "Incorrect Outputs generated in TEST 1, Phase 1."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 1, PHASE 1.                ";
Print_Message (Message);
End_Message;

-- TEST 1, Phase 2
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch  <= Heat;
Motor_Speed    <= Threshold;
Water_Temp     <= Below_Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = On_State    ) and
    (On_Off_Indicator   = On_State    ) and
    (Ignition           = On_State    ) and
    (Oil_Valve_Control  = Open_State   ) and
    (Water_Valve_Control = Closed_State)
REPORT "Incorrect Outputs generated in TEST 1, Phase 2."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 1, PHASE 2.                ";
Print_Message (Message);
End_Message;

-- TEST 1, Phase 3
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch  <= Heat;
Motor_Speed    <= Threshold;
Water_Temp     <= Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = On_State    ) and
    (On_Off_Indicator   = On_State    ) and
    (Ignition           = On_State    ) and
    (Oil_Valve_Control  = Open_State   ) and
    (Water_Valve_Control = Open_State  )

```

```
REPORT "Incorrect Outputs generated in TEST 1, Phase 3."
SEVERITY Error;
```

```
Start_Message;
Message := "Passed TEST 1, PHASE 3.                ";
Print_Message (Message);
End_Message;
```

```
WAIT for 1 sec;
```

```
-- TEST 2, Phase 1
```

```
Start_Message;
Message := "TEST CASE 2: Normal Shut Off conditions are tested. There ";
Print_Message (Message);
Message := "are three phases in the test. Requirement R5 is tested.    ";
Print_Message (Message);
End_Message;
```

```
Air_Temp      <= 80;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow      <= Safe;
Master_Switch   <= Heat;
Motor_Speed     <= Threshold;
Water_Temp      <= Threshold;
```

```
WAIT for 1 sec;
```

```
ASSERT
```

```
(Motor_Control      = On_State ) and
(On_Off_Indicator   = On_State ) and
(Ignition           = On_State ) and
(Oil_Valve_Control  = Closed_State) and
(Water_Valve_Control = Open_State )
```

```
REPORT "Incorrect Outputs generated in TEST 2, Phase 1."
SEVERITY Error;
```

```
Start_Message;
Message := "Passed TEST 2, PHASE 1.                ";
Print_Message (Message);
End_Message;
```

```
-- TEST 2, Phase 2
```

```
Air_Temp      <= 80;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow      <= Safe;
Master_Switch   <= Heat;
Motor_Speed     <= Below_Threshold;
Water_Temp      <= Threshold;
```

```

WAIT for 5 sec;

ASSERT
    (Motor_Control      = Off_State  ) and
    (On_Off_Indicator   = Off_State  ) and
    (Ignition           = Off_State  ) and
    (Oil_Valve_Control  = Closed_State) and
    (Water_Valve_Control = Open_State )
REPORT "Incorrect Outputs generated in TEST 2, Phase 2."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 2, PHASE 2.          ";
Print_Message (Message);
End_Message;

-- TEST 2, Phase 3
Air_Temp      <= 80;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow      <= Safe;
Master_Switch   <= Heat;
Motor_Speed     <= Below_Threshold;
Water_Temp      <= Below_Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = Off_State  ) and
    (On_Off_Indicator   = Off_State  ) and
    (Ignition           = Off_State  ) and
    (Oil_Valve_Control  = Closed_State) and
    (Water_Valve_Control = Closed_State)
REPORT "Incorrect Outputs generated in TEST 2, Phase 3."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 2, PHASE 3.          ";
Print_Message (Message);
End_Message;

WAIT for 5 min;

-- RESTART THE HEATER SYSTEM PRIOR TO NEXT TEST (Run Test 1 Again).
-- Restart, Phase 1
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow      <= Safe;
Master_Switch   <= Heat;
Motor_Speed     <= Below_Threshold;

```

```

Water_Temp          <= Below_Threshold;

WAIT for 1 sec;

-- Restart, Phase 2
Air_Temp             <= 75;
Desired_Air_Temp     <= 78;
Combustion_Sensor    <= Safe;
Fuel_Flow            <= Safe;
Master_Switch        <= Heat;
Motor_Speed          <= Threshold;
Water_Temp           <= Below_Threshold;

WAIT for 1 sec;

-- Restart, Phase 3
Air_Temp             <= 75;
Desired_Air_Temp     <= 78;
Combustion_Sensor    <= Safe;
Fuel_Flow            <= Safe;
Master_Switch        <= Heat;
Motor_Speed          <= Threshold;
Water_Temp           <= Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control    = On_State    ) and
    (On_Off_Indicator = On_State    ) and
    (Ignition         = On_State    ) and
    (Oil_Valve_Control = Open_State ) and
    (Water_Valve_Control = Open_State )
REPORT "System did not start correctly before Test 3, Combustion Error."
SEVERITY Error;

Start_Message;
Message := "System Restart before Test 3.                ";
Print_Message (Message);
End_Message;

WAIT for 1 sec;

-- TEST 3, Combustion Error
Start_Message;
Message := "TEST CASE 3: This test case covers shut downs as a result of";
Print_Message (Message);
Message := "error conditions. The first phase tests a Combustion Sensor";
Print_Message (Message);
Message := "error as defined in Requirement R4.          ";
Print_Message (Message);
End_Message;

```

```

Air_Temp          <= 75;
Desired_Air_Temp  <= 78;
Combustion_Sensor <= UnSafe;
Fuel_Flow         <= Safe;
Master_Switch     <= Heat;
Motor_Speed       <= Threshold;
Water_Temp        <= Threshold;

```

WAIT for 6 sec;

```

ASSERT
    (Motor_Control      = Off_State  ) and
    (On_Off_Indicator   = Off_State  ) and
    (Ignition           = Off_State  ) and
    (Oil_Valve_Control  = Closed_State) and
    (Water_Valve_Control = Open_State )
REPORT "Incorrect Outputs generated in TEST 3, Combustion Error."
SEVERITY Error;

```

```

Start_Message;
Message := "Passed TEST 3, Combustion Error Test.      ";
Print_Message (Message);
End_Message;

```

WAIT for 5 min;

-- RESTART THE HEATER SYSTEM PRIOR TO NEXT TEST (Run Test 1 Again).

```

-- Restart, Phase 1
Air_Temp          <= 75;
Desired_Air_Temp  <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow         <= Safe;
Master_Switch     <= Heat;
Motor_Speed       <= Below_Threshold;
Water_Temp        <= Below_Threshold;

```

WAIT for 1 sec;

```

-- Restart, Phase 2
Air_Temp          <= 75;
Desired_Air_Temp  <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow         <= Safe;
Master_Switch     <= Heat;
Motor_Speed       <= Threshold;
Water_Temp        <= Below_Threshold;

```

WAIT for 1 sec;

```

-- Restart, Phase 3
Air_Temp          <= 75;
Desired_Air_Temp  <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow         <= Safe;
Master_Switch     <= Heat;
Motor_Speed       <= Threshold;
Water_Temp        <= Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = On_State    ) and
    (On_Off_Indicator   = On_State    ) and
    (Ignition           = On_State    ) and
    (Oil_Valve_Control  = Open_State   ) and
    (Water_Valve_Control = Open_State   )
REPORT "System did not start correctly before Test 3, Fuel Flow Error."
SEVERITY Error;

Start_Message;
Message := "System Restart before TEST 3, Fuel Flow Error.          ";
Print_Message (Message);
End_Message;

WAIT for 10 sec;

-- TEST 3, Fuel Flow Error
Start_Message;
Message := "TEST CASE 3: This test case covers shut downs as a result of";
Print_Message (Message);
Message := "error conditions. This second phase tests a Fuel Flow error";
Print_Message (Message);
Message := "as defined in Requirement R4.                          ";
Print_Message (Message);
End_Message;

Air_Temp          <= 75;
Desired_Air_Temp  <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow         <= Unsafe;
Master_Switch     <= Heat;
Motor_Speed       <= Threshold;
Water_Temp        <= Threshold;

WAIT for 6 sec;

ASSERT
    (Motor_Control      = Off_State    ) and
    (On_Off_Indicator   = Off_State    ) and
    (Ignition           = Off_State    ) and

```

```

        (Oil_Valve_Control = Closed_State) and
        (Water_Valve_Control = Open_State )
    REPORT "Incorrect Outputs generated in TEST 3, Fuel Flow Error."
    SEVERITY Error;

Start_Message;
Message := "Passed TEST 3, Fuel Flow Error Test.           ";
Print_Message (Message);
End_Message;

    WAIT for 5 min;

-- RESTART THE HEATER SYSTEM PRIOR TO NEXT TEST (Run Test 1 Again).
    -- Restart, Phase 1
    Air_Temp      <= 75;
    Desired_Air_Temp <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow      <= Safe;
    Master_Switch   <= Heat;
    Motor_Speed     <= Below_Threshold;
    Water_Temp      <= Below_Threshold;

    WAIT for 1 sec;

    -- Restart, Phase 2
    Air_Temp      <= 75;
    Desired_Air_Temp <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow      <= Safe;
    Master_Switch   <= Heat;
    Motor_Speed     <= Threshold;
    Water_Temp      <= Below_Threshold;

    WAIT for 1 sec;

    -- Restart, Phase 3
    Air_Temp      <= 75;
    Desired_Air_Temp <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow      <= Safe;
    Master_Switch   <= Heat;
    Motor_Speed     <= Threshold;
    Water_Temp      <= Threshold;

    WAIT for 1 sec;

    ASSERT
        (Motor_Control      = On_State ) and
        (On_Off_Indicator    = On_State ) and
        (Ignition            = On_State ) and
        (Oil_Valve_Control    = Open_State ) and

```



```

        (Water_Valve_Control = Open_State )
    REPORT "System did not start correctly before Test 4."
    SEVERITY Error;

Start_Message;
Message := "System Restart before Test 4.                ";
Print_Message (Message);
End_Message;

    WAIT for 10 sec;

    -- TEST 4, Phase 1
Start_Message;
Message := "TEST CASE 4: Shut Down due to Master Switch turned off. ";
Print_Message (Message);
Message := "There are three phases in this test. Requirement R5 and ";
Print_Message (Message);
Message := "Constraint C2 are tested in this test.                ";
Print_Message (Message);
End_Message;

    Air_Temp          <= 75;
    Desired_Air_Temp  <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow         <= Safe;
    Master_Switch     <= Off;
    Motor_Speed       <= Threshold;
    Water_Temp        <= Threshold;

    WAIT for 1 sec;

    ASSERT
        (Motor_Control      = On_State    ) and
        (On_Off_Indicator   = On_State    ) and
        (Ignition           = On_State    ) and
        (Oil_Valve_Control  = Closed_State) and
        (Water_Valve_Control = Open_State )
    REPORT "Incorrect Outputs generated in TEST 4, Phase 1."
    SEVERITY Error;

Start_Message;
Message := "Passed TEST 4, PHASE 1.                ";
Print_Message (Message);
End_Message;

    -- TEST 4, Phase 2
    Air_Temp          <= 75;
    Desired_Air_Temp  <= 78;
    Combustion_Sensor <= Safe;
    Fuel_Flow         <= Safe;
    Master_Switch     <= Off;

```

```

Motor_Speed      <= Below_Threshold;
Water_Temp       <= Threshold;

WAIT for 5 sec;

ASSERT
    (Motor_Control      = Off_State ) and
    (On_Off_Indicator   = Off_State ) and
    (Ignition           = Off_State ) and
    (Oil_Valve_Control  = Closed_State) and
    (Water_Valve_Control = Open_State )
REPORT "Incorrect Outputs generated in TEST 4, Phase 2."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 4, PHASE 2.                ";
Print_Message (Message);
End_Message;

-- TEST 4, Phase 3
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch <= Off;
Motor_Speed   <= Below_Threshold;
Water_Temp    <= Below_Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = Off_State ) and
    (On_Off_Indicator   = Off_State ) and
    (Ignition           = Off_State ) and
    (Oil_Valve_Control  = Closed_State) and
    (Water_Valve_Control = Closed_State)
REPORT "Incorrect Outputs generated in TEST 4, Phase 3."
SEVERITY Error;

Start_Message;
Message := "Passed TEST 4, PHASE 3.                ";
Print_Message (Message);
End_Message;

WAIT for 1 min;

-- TEST 5
Start_Message;
Message := "TEST CASE 5: Attempt to restart the Heating System after ";
Print_Message (Message);
Message := "only one (1) minute has expired. The System should not ";

```

```

Print_Message (Message);
Message := "start until after 5 minutes has expired. Tests Constraint 1";
Print_Message (Message);
End_Message;

-- Restart, Phase 1
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch <= Heat;
Motor_Speed   <= Below_Threshold;
Water_Temp    <= Below_Threshold;

WAIT for 1 sec;

-- Restart, Phase 2
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch <= Heat;
Motor_Speed   <= Threshold;
Water_Temp    <= Below_Threshold;

WAIT for 1 sec;

-- Restart, Phase 3
Air_Temp      <= 75;
Desired_Air_Temp <= 78;
Combustion_Sensor <= Safe;
Fuel_Flow     <= Safe;
Master_Switch <= Heat;
Motor_Speed   <= Threshold;
Water_Temp    <= Threshold;

WAIT for 1 sec;

ASSERT
    (Motor_Control      = On_State    ) and
    (On_Off_Indicator   = On_State    ) and
    (Ignition           = On_State    ) and
    (Oil_Valve_Control  = Open_State   ) and
    (Water_Valve_Control = Open_State   )
REPORT "TRIED TO START SYSTEM BEFORE 5 MIN TIME HAD ELAPSED."
SEVERITY Note;

Start_Message;
Message := "TEST CASE 5, Phase 2: Wait an additional 4 minutes and test";
Print_Message (Message);
Message := "if the system will restart after the total 5 minutes has ";

```

```

Print_Message (Message);
Message := "expired.                                     ";
Print_Message (Message);
End_Message;

WAIT for 4 min;

ASSERT
    (Motor_Control      = On_State    ) and
    (On_Off_Indicator   = On_State    ) and
    (Ignition           = On_State    ) and
    (Oil_Valve_Control  = Open_State  ) and
    (Water_Valve_Control = Open_State  )
REPORT "System did not start correctly in Test 5."
SEVERITY Error;

Start_Message;
Message := "System Restart Correctly in Test 5 after 5 min time expired.";
Print_Message (Message);
End_Message;

Start_Message;
End_Message;

-- END THE SIMULATION
    ASSERT false
    REPORT "HEATER SYSTEM SIMULATION PASSED TESTING"
    SEVERITY ERROR;

    end process stimulus;
end bench;

```

C.15 Heater System Configuration File

```
1  --*****
2  --*
3  --*  TITLE:      Heating System Problem
4  --*  DATE:       11 Aug 91
5  --*  VERSION:    1.0
6  --*  FILENAME:   HEATER_SYSTEM_CONFIG.VHD
7  --*
8  --*  FUNCTION:   Configure the components for test.
9  --*
10 --*  AUTHOR:     Capt Dan Eickmeier
11 --*  HISTORY:    V1.0 - 10 Aug 91 - Initial Version
12 --*              - 11 Aug 91 - Compiled Successfully
13 --*
14 --*****

15 configuration Struct_Config of system is
16     use Work.all;
17     for structure
18         for all: Control_Heater_UUT
19             use entity Work.Control_Heater (Structure);
20             for structure
21                 for all: Shut_Off_Furnace
22                     use entity Work.Shut_Off_Furnace (Behavior);
23                 end for;

24             for all: Control_Motor
25                 use entity Work.Control_Motor (Structure);
26                 for structure
27                     for all: Compare_Temperature
28                         use entity Work.Compare_Temperature (Behavior);
29                     end for;
30                     for all: Activate_Motor
31                         use entity Work.Activate_Motor (Behavior);
32                     end for;
33                 end for;
34             end for;

35         for all: Control_Furnace_Ignition
36             use entity Work.Control_Furnace_Ignition (Structure);
37             for structure
38                 for all: Monitor_Motor_Speed
39                     use entity Work.Monitor_Motor_Speed (Behavior);
40                 end for;
41                 for all: Activate_Ignition
42                     use entity Work.Activate_Ignition (Behavior);
43                 end for;
44                 for all: Activate_Oil_Valve
45                     use entity Work.Activate_Oil_Valve (Behavior);
46                 end for;
47             end for;
```

```

48             end for;

49             for all: Control_Water_Valve
50                 use entity Work.Control_Water_Valve (Behavior);
51             end for;

52         end for;
53     end for;

54     for all: Delay_Five_Min_UUT
55         use entity Work.Delay_Five_Min (Behavior);
56     end for;

57     for all: Delay_Five_Sec_UUT
58         use entity Work.Delay_Five_Sec (Behavior);
59     end for;

60     for all: testbench
61         use entity Work.test ( Bench );
62     end for;
63     end for;
64 end;

```

C.16 Heater System Compilation Script

```
1  #!/bin/csh
2  # COMPILE.COM
3  #   This is a UNIX script file for compiling (analyzing)
4  #   the Heating System Problem in the ZYCAD VHDL Environment.
5  #
6  #
7  zvan -nc -l types.vhd
8  zvan -nc -l heat_a-0.vhd
9  zvan -nc -l heat_a0.vhd
10 zvan -nc -l heat_a2.vhd
11 zvan -nc -l heat_a3.vhd
12 zvan -nc -l timers.vhd
13 #
14 zvan -nc -l heat_a3_behav.vhd
15 zvan -nc -l heat_a2_behav.vhd
16 zvan -nc -l heat_a0_behav.vhd
17 zvan -nc -l heat_a0_struct.vhd
18 zvan -nc -l heat_a-0_struct.vhd
19 zvan -nc -l timers_behav.vhd
20 #
21 zvan -nc -l testbench.vhd
22 zvan -nc -l heater_system.vhd
23 zvan -nc -l heater_system_config.vhd
24 #
25 # END of Script
```

C.17 Heater System Simulation Script

```
1  #! /bin/csh
2  # GOHEAT.COM
3  #   This is a UNIX script file for running the
4  #   Heating System Problem in the ZYCAD VHDL simulator.
5  #   The command line 'includes' a file heat_batch.inc which
6  #   contains simulator commands. The output of the simulation
7  #   is written to the file heat.output.
8  #
9  #   zvsim -nc -i heat_batch.inc -o heat.output struct_config
10 #
11 # END OF SCRIPT
```


C.18 Heater System Simulation Include File

```
1  --*****
2  -- Command File to include when running the Heater Problem in BATCH MODE
3  --*****
4  cd
5  echo "*****"
6  echo Simulation of the Heater System specified in VHDL
7  echo "*****"
8  echo
9  echo INITIAL VALUES ARE:
10 echo
11 cd system
12 fpr "%10t          = %r\n" AIR_TEMP I1
13 fpr "%18t          = %r\n" DESIRED_AIR_TEMP C3
14 fpr "%19t          = %r\n" COMBUSTION_SENSOR C1
15 fpr "%11t          = %r\n" FUEL_FLOW C2
16 fpr "%15t          = %r\n" MASTER_SWITCH C4
17 fpr "%16t          = %r\n" FIVE_SEC_TIMER C5
18 fpr "%16t          = %r\n" FIVE_MIN_TIMER C6
19 fpr "%13t          = %r\n" MOTOR_SPEED C7
20 fpr "%12t          = %r\n" WATER_TEMP C8
21 fpr "%15t          = %r\n" MOTOR_CONTROL O1
22 fpr "%18t          = %r\n" ON/OFF_INDICATOR O2
23 fpr "%10t          = %r\n" IGNITION O3
24 fpr "%19t          = %r\n" OIL_VALVE_CONTROL O4
25 fpr "%21t          = %r\n" WATER_VALVE_CONTROL O5
26 echo
27 --
28 monitor -n I1 event /system/I1
29 edit I1
30 fpr "%10t          = %r\n" AIR_TEMP I1
31 end --edit
32 --
33 monitor -n C3 event /system/C3
34 edit C3
35 fpr "%18t          = %r\n" DESIRED_AIR_TEMP C3
36 end --edit
37 --
38 monitor -n C1 event /system/C1
39 edit C1
40 fpr "%19t          = %r\n" COMBUSTION_SENSOR C1
41 end --edit
42 --
43 monitor -n C2 event /system/C2
44 edit C2
45 fpr "%11t          = %r\n" FUEL_FLOW C2
46 end --edit
47 --
48 monitor -n C4 event /system/C4
49 edit C4
50 fpr "%15t          = %r\n" MASTER_SWITCH C4
```

```

51 end --edit
52 --
53 monitor -n C5 event /system/C5
54 edit C5
55 fpr "%16t      = %r\n" FIVE_SEC_TIMER C5
56 end --edit
57 --
58 monitor -n C6 event /system/C6
59 edit C6
60 fpr "%16t      = %r\n" FIVE_MIN_TIMER C6
61 end --edit
62 --
63 monitor -n C7 event /system/C7
64 edit C7
65 fpr "%13t      = %r\n" MOTOR_SPEED C7
66 end --edit
67 --
68 monitor -n C8 event /system/C8
69 edit C8
70 fpr "%12t      = %r\n" WATER_TEMP C8
71 end --edit
72 --
73 monitor -n O1 event /system/O1
74 edit O1
75 fpr "%15t      = %r\n" MOTOR_CONTROL O1
76 end --edit
77 --
78 monitor -n O2 event /system/O2
79 edit O2
80 fpr "%18t      = %r\n" ON/OFF_INDICATOR O2
81 end --edit
82 --
83 monitor -n O3 event /system/O3
84 edit O3
85 fpr "%10t      = %r\n" IGNITION O3
86 end --edit
87 --
88 monitor -n O4 event /system/O4
89 edit O4
90 fpr "%19t      = %r\n" OIL_VALVE_CONTROL O4
91 end --edit
92 --
93 monitor -n O5 event /system/O5
94 edit O5
95 fpr "%21t = %r\n" WATER_VALVE_CONTROL O5
96 end --edit
97 --
98 run
99 quit

```

C.19 Heater System Simulation Output

```
"*****"  
Simulation of the Heater System specified in VHDL  
"*****"
```

INITIAL VALUES ARE:

```
AIR_TEMP           = -2147483648  
DESIRED_AIR_TEMP    = -2147483648  
COMBUSTION_SENSOR   = SAFE  
FUEL_FLOW           = SAFE  
MASTER_SWITCH       = HEAT  
FIVE_SEC_TIMER       = TRUE  
FIVE_MIN_TIMER       = TRUE  
MOTOR_SPEED         = BELOW_THRESHOLD  
WATER_TEMP          = BELOW_THRESHOLD  
MOTOR_CONTROL       = OFF_STATE  
ON/OFF_INDICATOR    = OFF_STATE  
IGNITION            = 3  
OIL_VALVE_CONTROL   = 4  
WATER_VALVE_CONTROL = CLOSED_STATE
```

```
*****  
TEST CASE 1: Initial Startup conditions tested. There  
are three phases in the test. Requirements R1, R2, and R3  
are tested.  
*****
```

0 SEC

```
DESIRED_AIR_TEMP    = 78  
AIR_TEMP            = 75  
MOTOR_CONTROL       = ON_STATE  
ON/OFF_INDICATOR    = ON_STATE  
FIVE_MIN_TIMER       = FALSE
```

```
*****  
Passed TEST 1, PHASE 1.  
*****
```

1 SEC

```
MOTOR_SPEED         = THRESHOLD  
IGNITION            = 3  
OIL_VALVE_CONTROL   = 4  
FIVE_SEC_TIMER       = FALSE
```

```
*****  
Passed TEST 1, PHASE 2.  
*****
```

2 SEC

WATER_TEMP = THRESHOLD
WATER_VALVE_CONTROL = OPEN_STATE

Passed TEST 1, PHASE 3.

TEST CASE 2: Normal Shut Off conditions are tested. There
are three phases in the test. Requirement R5 is tested.

4 SEC
AIR_TEMP = 80
OIL_VALVE_CONTROL = 4

Passed TEST 2, PHASE 1.

5 SEC
MOTOR_SPEED = BELOW_THRESHOLD
IGNITION = 3
9 SEC
FIVE_SEC_TIMER = TRUE
MOTOR_CONTROL = OFF_STATE
ON/OFF_INDICATOR = OFF_STATE

Passed TEST 2, PHASE 2.

10 SEC
WATER_TEMP = BELOW_THRESHOLD
WATER_VALVE_CONTROL = CLOSED_STATE

Passed TEST 2, PHASE 3.

309 SEC
FIVE_MIN_TIMER = TRUE
311 SEC
AIR_TEMP = 75
MOTOR_CONTROL = ON_STATE
ON/OFF_INDICATOR = ON_STATE
FIVE_MIN_TIMER = FALSE
312 SEC
MOTOR_SPEED = THRESHOLD
IGNITION = 3

```

OIL_VALVE_CONTROL = 4
FIVE_SEC_TIMER    = FALSE
313 SEC
WATER_TEMP        = THRESHOLD
WATER_VALVE_CONTROL = OPEN_STATE

```

```

*****
System Restart before Test 3.
*****

```

```

*****
TEST CASE 3: This test case covers shut downs as a result of
error conditions. The first phase tests a Combustion Sensor
error as defined in Requirement R4.
*****

```

```

315 SEC
COMBUSTION_SENSOR = UNSAFE
IGNITION          = 3
OIL_VALVE_CONTROL = 4
320 SEC
FIVE_SEC_TIMER    = TRUE
MOTOR_CONTROL     = OFF_STATE
ON/OFF_INDICATOR  = OFF_STATE

```

```

*****
Passed TEST 3, Combustion Error Test.
*****

```

```

620 SEC
FIVE_MIN_TIMER    = TRUE
621 SEC
WATER_TEMP        = BELOW_THRESHOLD
MOTOR_SPEED       = BELOW_THRESHOLD
COMBUSTION_SENSOR = SAFE
WATER_VALVE_CONTROL = CLOSED_STATE
IGNITION          = 3
OIL_VALVE_CONTROL = 4
MOTOR_CONTROL     = ON_STATE
ON/OFF_INDICATOR  = ON_STATE
FIVE_SEC_TIMER    = FALSE
FIVE_MIN_TIMER    = FALSE
IGNITION          = 3
OIL_VALVE_CONTROL = 4
622 SEC
MOTOR_SPEED       = THRESHOLD
IGNITION          = 3
OIL_VALVE_CONTROL = 4
623 SEC
WATER_TEMP        = THRESHOLD

```

WATER_VALVE_CONTROL = OPEN_STATE

System Restart before TEST 3, Fuel Flow Error.

TEST CASE 3: This test case covers shut downs as a result of
error conditions. This second phase tests a Fuel Flow error
as defined in Requirement R4.

634 SEC

FUEL_FLOW = UNSAFE
IGNITION = 3
OIL_VALVE_CONTROL = 4

639 SEC

FIVE_SEC_TIMER = TRUE
MOTOR_CONTROL = OFF_STATE
ON/OFF_INDICATOR = OFF_STATE

Passed TEST 3, Fuel Flow Error Test.

939 SEC

FIVE_MIN_TIMER = TRUE

940 SEC

WATER_TEMP = BELOW_THRESHOLD
MOTOR_SPEED = BELOW_THRESHOLD
FUEL_FLOW = SAFE
WATER_VALVE_CONTROL = CLOSED_STATE
IGNITION = 3
OIL_VALVE_CONTROL = 4
MOTOR_CONTROL = ON_STATE
ON/OFF_INDICATOR = ON_STATE
FIVE_SEC_TIMER = FALSE
FIVE_MIN_TIMER = FALSE
IGNITION = 3
OIL_VALVE_CONTROL = 4

941 SEC

MOTOR_SPEED = THRESHOLD
IGNITION = 3
OIL_VALVE_CONTROL = 4

942 SEC

WATER_TEMP = THRESHOLD
WATER_VALVE_CONTROL = OPEN_STATE

System Restart before Test 4.

TEST CASE 4: Shut Down due to Master Switch turned off.
There are three phases in this test. Requirement R5 and
Constraint C2 are tested in this test.

953 SEC

MASTER_SWITCH = OFF
OIL_VALVE_CONTROL = 4

Passed TEST 4, PHASE 1.

954 SEC

MOTOR_SPEED = BELOW_THRESHOLD
IGNITION = 3

958 SEC

FIVE_SEC_TIMER = TRUE
MOTOR_CONTROL = OFF_STATE
ON/OFF_INDICATOR = OFF_STATE

Passed TEST 4, PHASE 2.

959 SEC

WATER_TEMP = BELOW_THRESHOLD
WATER_VALVE_CONTROL = CLOSED_STATE

Passed TEST 4, PHASE 3.

TEST CASE 5: Attempt to restart the Heating System after
only one (1) minute has expired. The System should not
start until after 5 minutes has expired. Tests Constraint 1

1020 SEC

MASTER_SWITCH = HEAT

1021 SEC

MOTOR_SPEED = THRESHOLD

1022 SEC

WATER_TEMP = THRESHOLD
WATER_VALVE_CONTROL = OPEN_STATE

1023 SEC

Assertion NOTE at 1023 SEC in design unit BENCH from process /SYSTEM/BENCH/STIMULUS:
"TRIED TO START SYSTEM BEFORE 5 MIN TIME HAD ELAPSED."

TEST CASE 5 Phase 2: Wait an additional 4 minutes and test
if the system will restart after the total 5 minutes has
expired.

1258 SEC

FIVE_MIN_TIMER = TRUE
MOTOR_CONTROL = ON_STATE
ON/OFF_INDICATOR = ON_STATE
FIVE_MIN_TIMER = FALSE
IGNITION = 3
OIL_VALVE_CONTROL = 4
FIVE_SEC_TIMER = FALSE

System Restart Correctly in Test 5 after 5 min time expired.

1263 SEC

Assertion ERROR at 1263 SEC in design unit BENCH from process /SYSTEM/BENCH/STIMULUS:
"HEATER SYSTEM SIMULATION PASSED TESTING"

Appendix D. *The Lift Control System Problem Source Code*

Appendix D contains the source code for the Lift Control System. This Appendix is maintained as a separate document at the Air Force Institute of Technology, Department of Electrical and Computer Engineering, Wright-Patterson AFB, OH 45433. The points of contact are Maj K. Kanzaki or Maj P. Bailor. They may be reached at Commercial: (513)255-3576 or DSN: 785-3576.

Appendix E. Lift Control System Simulation Output

E.1 Lift Control System Testbench Entity and Architecture Declarations

```
--*****
--*
--* TITLE:      Lift Control System
--* DATE:       20 Sep 91
--* VERSION:    1.4
--* FILENAME:   TESTBENCH.VHD
--*
--* FUNCTION:    Entity and Behavior description for the Lift Control
--*              System Testbench.
--*
--* AUTHOR:      Capt Dan Eickmeier
--* HISTORY:     V1.0 - 30 Aug 91 - Initial Version
--*              30 Aug 91 - Compiles Successfully
--*              V1.1 - 06 Sep 91 - Added additional tests of functions.
--*              06 Sep 91 - Compiles Successfully
--*              V1.2 - 17 Sep 91 - Added Test 6
--*              17 Sep 91 - Compiles Successfully
--*              V1.3 - 18 Sep 91 - Added TEXTIO output.
--*              18 Sep 91 - Compiles Successfully
--*              V1.4 - 20 Sep 91 - Added Tests 7,8, and 9
--*              20 Sep 91 - Compiles Successfully
--*
--*****

use Work.Lift_Package.all;

entity test is
    port ( Emer_Button : out Emer_Button_Type := (False,False);
          Sum_Button   : out Sum_Button_Type  := (0,Nil);
          Dest_Button  : out Dest_Button_Type := (0,0);
          Light_On     : in Light_On_Type;
          Light_Off    : in Light_Off_Type;
          Emer_Signal   : in Emer_Signal_Type;
          Motor_Ind     : in Motor_Ind_Type;
          Lift_Status   : in Lift_Status_Type);
end test;

use Std.TextIO;
architecture bench of test is

begin
    stimulus: process

        subtype Message_Type is String (1 to 60);
        variable Message      : Message_Type := (others => ' ');
        variable NullString    : Message_Type := (others => ' ');
        variable StarString    : Message_Type := (others => '*');
        variable outline       : TextIO.Line;
        variable null_line     : TextIO.Line;
```

```

variable star_line : TextIO.Line;

procedure Start_Message is
begin
    TextIO.Write(null_line, NullString);
    TextIO.Write(star_line, StarString);
    TextIO.WriteLine(TextIO.OUTPUT, null_line);
    TextIO.WriteLine(TextIO.OUTPUT, star_line);
end Start_Message;

procedure End_Message is
begin
    TextIO.Write(null_line, NullString);
    TextIO.Write(star_line, StarString);
    TextIO.WriteLine(TextIO.OUTPUT, star_line);
    TextIO.WriteLine(TextIO.OUTPUT, null_line);
end End_Message;

procedure Print_Message ( In_Message : Message_Type ) is
begin
    TextIO.Write(outline, In_Message);
    TextIO.WriteLine(TextIO.OUTPUT, outline);
end Print_Message;

begin

    -- TEST CASE 1
    Start_Message;
    Message := "TEST CASE 1: Two Lifts Moving with no Error Conditions. ";
    Print_Message (Message);
    Message := "This test demonstrates design compliance with Problem ";
    Print_Message (Message);
    Message := "Constraints 1, 2(a), 3 and 4. ";
    Print_Message (Message);
    End_Message;

    Emer_Button.L1_Value <= False;
    Emer_Button.L2_Value <= False;

    Start_Message;
    Message := "Test 1: 2 UP SUMMONS REQUEST. ";
    Print_Message (Message);
    End_Message;

    Sum_Button.Floor <= 2;
    Sum_Button.Dir <= Up;
    Dest_Button.Floor <= 0;
    Dest_Button.Lift <= 0;

    WAIT for 1 sec;

    Start_Message;
    Message := "Test 1: 3 DOWN SUMMONS REQUEST. ";
    Print_Message (Message);
    End_Message;

```

```

Sum_Button.Floor    <= 3,      0 after 1 sec;
Sum_Button.Dir      <= Down,   Nil after 1 sec;

WAIT for 10 sec;

Start_Message;
Message := "Test 1:  DESTINATION FLOOR 3 ON LIFT 1.           ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 3,      0 after 1 sec;
Dest_Button.Lift    <= 1,      0 after 1 sec;

WAIT for 11 sec;

Start_Message;
Message := "Test 1:  DESTINATION FLOOR 1 ON LIFT 2.           ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 1,      0 after 1 sec;
Dest_Button.Lift    <= 2,      0 after 1 sec;

WAIT for 78 sec;

-- TEST 2
Start_Message;
Message := "TEST CASE 2: Lift 1 Moving; Lift 2 in Emergency Condition.  ";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem        ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 3, 5 and 6.                            ";
Print_Message (Message);
Message := "Note: The 2 UP SUMMONS occurs as the lift is going DOWN to   ";
Print_Message (Message);
Message := "meet the DESTINATION request, so the lift does not stop.     ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 2:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

Emer_Button.L1_Value <= False;
Emer_Button.L2_Value <= True;
Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- Simulation Time of 1 sec

Start_Message;

```

```

Message := "Test 2: 3 DOWN SUMMONS REQUEST.                ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 3,      0 after 1 sec;
Sum_Button.Dir      <= Down,   Nil after 1 sec;

WAIT for 1 sec;  -- Simulation Time of 2 sec

Start_Message;
Message := "Test 2: DESTINATION FLOOR 1 ON LIFT 1          ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 1,      0 after 1 sec;
Dest_Button.Lift    <= 1,      0 after 1 sec;

WAIT for 1 sec;  -- Simulation Time of 3 sec

Start_Message;
Message := "Test 2: 2 UP SUMMONS REQUEST.                  ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 2,      0 after 1 sec;
Sum_Button.Dir      <= Up,     Nil after 1 sec;

WAIT for 2 sec;  -- Simulation Time of 5 sec

Start_Message;
Message := "Test 2: DESTINATION FLOOR 1 ON LIFT 2.         ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 1,      0 after 1 sec;
Dest_Button.Lift    <= 2,      0 after 1 sec;

WAIT for 95 sec;

-- TEST 3
Start_Message;
Message := "TEST CASE 3: Both Lift 1 and Lift 2 in Emergency Conditions.";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem      ";
Print_Message (Message);
Message := "Constraint 6 and a design decision to disregard Summons    ";
Print_Message (Message);
Message := "Requests when both Lifts are Out of Service.              ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 3: EMERGENCY BUTTONS ON BOTH LIFT 1 AND LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

```

```

Emer_Button.L1_Value <= True;
Emer_Button.L2_Value <= True;
Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 3:  3 DOWN SUMMONS REQUEST.                ";
Print_Message (Message);
Message := "Test 3:  Note that no lights are illuminated.    ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 3,      0 after 1 sec;
Sum_Button.Dir      <= Down, Nil after 1 sec;

WAIT for 99 sec;

-- TEST 4
Start_Message;
Message := "TEST CASE 4: Lift 1 Moving; Lift 2 in Emergency Condition. ";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem      ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 3, 5, and 6.  Note that the 2 UP        ";
Print_Message (Message);
Message := "SUMMONS request occurs while going UP from floor 1, and it   ";
Print_Message (Message);
Message := "is handled enroute.                                         ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 4:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

Emer_Button.L1_Value <= False;
Emer_Button.L2_Value <= True;

Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 4:  2 DOWN SUMMONS REQUEST.                ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 2,      0 after 1 sec;

```

```

Sum_Button.Dir      <= Down, Nil after 1 sec;

WAIT for 1 sec;  -- 2 Sec Simulation Time

Start_Message;
Message := "Test 4: DESTINATION FLOOR 1 ON LIFT 1.           ";
Print_Message (Message);
End_Message;

Dest_Button.Floor    <= 1,      0 after 1 sec;
Dest_Button.Lift     <= 1,      0 after 1 sec;

WAIT for 1 sec;  -- 3 Sec Simulation Time

Start_Message;
Message := "Test 4: 3 DOWN SUMMONS REQUEST.                 ";
Print_Message (Message);
End_Message;

Sum_Button.Floor     <= 3,      0 after 1 sec;
Sum_Button.Dir       <= Down, Nil after 1 sec;

WAIT for 2 sec;  -- 5 Sec Simulation Time

Start_Message;
Message := "Test 4: 2 UP SUMMONS REQUEST.                   ";
Print_Message (Message);
End_Message;

Sum_Button.Floor     <= 2,      0 after 1 sec;
Sum_Button.Dir       <= Up,   Nil after 1 sec;

WAIT for 28 sec;  -- 33 Sec Simulation Time

Start_Message;
Message := "Test 4: DESTINATION FLOOR 1 ON LIFT 1.           ";
Print_Message (Message);
End_Message;

Dest_Button.Floor    <= 1,      0 after 1 sec;
Dest_Button.Lift     <= 1,      0 after 1 sec;

WAIT for 67 sec;  -- END of TEST 4

-- TEST 5
Start_Message;
Message := "TEST CASE 5: Both Lift 1 and Lift 2 in Service.  ";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem ";
Print_Message (Message);
Message := "Constraints 1, 2, and 3. Note that both requests given are ";
Print_Message (Message);
Message := "for the floor where the lifts are currently located and no ";
Print_Message (Message);
Message := "action is taken except illuminating and extinguishing lights";

```

```

Print_Message (Message);
End_Message;

    Emer_Button.L1_Value <= False;
    Emer_Button.L2_Value <= False;

    Sum_Button.Floor    <= 0;
    Sum_Button.Dir      <= Nil;
    Dest_Button.Floor   <= 0;
    Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 5:  1 UP SUMMONS REQUEST.          ";
Print_Message (Message);
End_Message;

    Sum_Button.Floor    <= 1,      0 after 1 sec;
    Sum_Button.Dir      <= Up,    Nil after 1 sec;

WAIT for 1 sec;  -- 2 Sec Simulation Time

Start_Message;
Message := "Test 5:  DESTINATION FLOOR 1 ON LIFT 2.          ";
Print_Message (Message);
End_Message;

    Dest_Button.Floor   <= 1,      0 after 1 sec;
    Dest_Button.Lift    <= 2,      0 after 1 sec;

WAIT for 98 sec;

    -- TEST 6
Start_Message;
Message := "TEST CASE 6: Lift 1 Running; Lift 2 in Emergency Conditions.";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem          ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 2(b), 3, 4, and 6.  Note the waiting          ";
Print_Message (Message);
Message := "time to handle both UP and DOWN requests on floor 2.          ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 6:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

    Emer_Button.L1_Value <= False;
    Emer_Button.L2_Value <= True;

    Sum_Button.Floor    <= 0;
    Sum_Button.Dir      <= Nil;
    Dest_Button.Floor   <= 0;

```



```

        Dest_Button.Lift      <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 6:  1 UP SUMMONS REQUEST.          ";
Print_Message (Message);
End_Message;

        Sum_Button.Floor      <= 1,      0 after 1 sec;
        Sum_Button.Dir         <= Up,    Nil after 1 sec;

WAIT for 1 sec;  -- 2 Sec Simulation Time

Start_Message;
Message := "Test 6:  DESTINATION FLOOR 3 ON LIFT 1.  ";
Print_Message (Message);
End_Message;

        Dest_Button.Floor      <= 3,      0 after 1 sec;
        Dest_Button.Lift       <= 1,      0 after 1 sec;

WAIT for 1 sec;  -- 3 Sec Simulation Time

Start_Message;
Message := "Test 6:  2 DOWN SUMMONS REQUEST.        ";
Print_Message (Message);
End_Message;

        Sum_Button.Floor      <= 2,      0 after 1 sec;
        Sum_Button.Dir         <= Down, Nil after 1 sec;

WAIT for 15 sec; -- 18 Sec Simulation Time

Start_Message;
Message := "Test 6:  2 UP SUMMONS REQUEST.          ";
Print_Message (Message);
End_Message;

        Sum_Button.Floor      <= 2,      0 after 1 sec;
        Sum_Button.Dir         <= Up,    Nil after 1 sec;

WAIT for 15 sec; -- 33 Sec Simulation Time

Start_Message;
Message := "Test 6:  DESTINATION FLOOR 1 ON LIFT 1.  ";
Print_Message (Message);
End_Message;

        Dest_Button.Floor      <= 1,      0 after 1 sec;
        Dest_Button.Lift       <= 1,      0 after 1 sec;

WAIT for 67 sec;  -- END of Test 6

-- TEST 7
Start_Message;

```

```

Message := "TEST CASE 7: Lift 1 Running; Lift 2 in Emergency Conditions.";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem      ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 2(b), 3, 4, and 6. Note the waiting  ";
Print_Message (Message);
Message := "time to handle both UP and DOWN requests on floor 2.      ";
Print_Message (Message);
End_Message;

```

```

Start_Message;
Message := "Test 7: LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

```

```

Emer_Button.L1_Value <= False;
Emer_Button.L2_Value <= True;

```

```

Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

```

WAIT for 1 sec; -- 1 Sec Simulation Time

```

Start_Message;
Message := "Test 7: 1 UP SUMMONS REQUEST.                               ";
Print_Message (Message);
End_Message;

```

```

Sum_Button.Floor    <= 1,      0 after 1 sec;
Sum_Button.Dir      <= Up,    Nil after 1 sec;

```

WAIT for 11 sec; -- 12 Sec Simulation Time

```

Start_Message;
Message := "Test 7: DESTINATION FLOOR 3 ON LIFT 1.                       ";
Print_Message (Message);
End_Message;

```

```

Dest_Button.Floor   <= 3,      0 after 1 sec;
Dest_Button.Lift    <= 1,      0 after 1 sec;

```

WAIT for 1 sec; -- 13 Sec Simulation Time

```

Start_Message;
Message := "Test 7: 2 DOWN SUMMONS REQUEST.                               ";
Print_Message (Message);
End_Message;

```

```

Sum_Button.Floor    <= 2,      0 after 1 sec;
Sum_Button.Dir      <= Down, Nil after 1 sec;

```

WAIT for 2 sec; -- 15 Sec Simulation Time

```

Start_Message;

```

```

Message := "Test 7:  2 UP SUMMONS REQUEST.                ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 2,      0 after 1 sec;
Sum_Button.Dir      <= Up,    Nil after 1 sec;

WAIT for 8 sec;  -- 23 Sec Simulation Time

Start_Message;
Message := "Test 7:  DESTINATION FLOOR 1 ON LIFT 1.        ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 1,      0 after 1 sec;
Dest_Button.Lift    <= 1,      0 after 1 sec;

WAIT for 77 sec;  -- END of Test 7

-- TEST 8
Start_Message;
Message := "TEST CASE 8: Lift 1 Running; Lift 2 in Emergency Conditions.";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem      ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 2(b), 3, 4, and 6.  Note the waiting  ";
Print_Message (Message);
Message := "time to handle both UP and DOWN requests on floor 2.      ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 8:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.";
Print_Message (Message);
End_Message;

Emer_Button.L1_Value <= False;
Emer_Button.L2_Value <= True;

Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 8:  1 UP SUMMONS REQUEST.                ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 1,      0 after 1 sec;
Sum_Button.Dir      <= Up,    Nil after 1 sec;

WAIT for 11 sec;  -- 12 Sec Simulation Time

```

```

Start_Message;
Message := "Test 8: DESTINATION FLOOR 2 ON LIFT 1.           ";
Print_Message (Message);
End_Message;

Dest_Button.Floor    <= 2,      0 after 1 sec;
Dest_Button.Lift     <= 1,      0 after 1 sec;

WAIT for 1 sec;  -- 13 Sec Simulation Time

Start_Message;
Message := "Test 8: 2 DOWN SUMMONS REQUEST.                 ";
Print_Message (Message);
End_Message;

Sum_Button.Floor     <= 2,      0 after 1 sec;
Sum_Button.Dir       <= Down, Nil after 1 sec;

WAIT for 2 sec;  -- 15 Sec Simulation Time

Start_Message;
Message := "Test 8: 2 UP SUMMONS REQUEST.                   ";
Print_Message (Message);
End_Message;

Sum_Button.Floor     <= 2,      0 after 1 sec;
Sum_Button.Dir       <= Up,   Nil after 1 sec;

WAIT for 8 sec;  -- 23 Sec Simulation Time

Start_Message;
Message := "Test 8: DESTINATION FLOOR 1 ON LIFT 1.         ";
Print_Message (Message);
End_Message;

Dest_Button.Floor    <= 1,      0 after 1 sec;
Dest_Button.Lift     <= 1,      0 after 1 sec;

WAIT for 77 sec;  -- END of Test 8

-- TEST 9
Start_Message;
Message := "TEST CASE 9: Both Lift 1 and Lift 2 Running.    ";
Print_Message (Message);
Message := "This test demonstrates design compliance with Problem ";
Print_Message (Message);
Message := "Constraints 1, 2(a), 2(b), 3, 4, and 6. Note the waiting ";
Print_Message (Message);
Message := "time to handle both UP and DOWN requests on floor 2. ";
Print_Message (Message);
End_Message;

Start_Message;
Message := "Test 9: LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 CLEARED.";
Print_Message (Message);
End_Message;

```

```

Emer_Button.L1_Value <= False;
Emer_Button.L2_Value <= False;

Sum_Button.Floor    <= 0;
Sum_Button.Dir      <= Nil;
Dest_Button.Floor   <= 0;
Dest_Button.Lift    <= 0;

WAIT for 1 sec;  -- 1 Sec Simulation Time

Start_Message;
Message := "Test 9:  1 UP SUMMONS REQUEST.      ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 1,      0 after 1 sec;
Sum_Button.Dir      <= Up,    Nil after 1 sec;

WAIT for 11 sec;  -- 12 Sec Simulation Time

Start_Message;
Message := "Test 9:  DESTINATION FLOOR 3 ON LIFT 1.      ";
Print_Message (Message);
End_Message;

Dest_Button.Floor   <= 3,      0 after 1 sec;
Dest_Button.Lift    <= 1,      0 after 1 sec;

WAIT for 1 sec;  -- 13 Sec Simulation Time

Start_Message;
Message := "Test 9:  2 DOWN SUMMONS REQUEST.      ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 2,      0 after 1 sec;
Sum_Button.Dir      <= Down, Nil after 1 sec;

WAIT for 2 sec;  -- 15 Sec Simulation Time

Start_Message;
Message := "Test 9:  2 UP SUMMONS REQUEST.      ";
Print_Message (Message);
End_Message;

Sum_Button.Floor    <= 2,      0 after 1 sec;
Sum_Button.Dir      <= Up,    Nil after 1 sec;

WAIT for 8 sec;  -- 23 Sec Simulation Time

Start_Message;
Message := "Test 9:  DESTINATION FLOOR 1 ON LIFT 1.      ";
Print_Message (Message);
End_Message;

```

```
Dest_Button.Floor    <= 1,    0 after 1 sec;  
Dest_Button.Lift     <= 1,    0 after 1 sec;
```

```
WAIT for 77 sec;  -- END of Test 9
```

```
    ASSERT false  
    REPORT "SIMULATION TESTING COMPLETE."  
    SEVERITY Error;  
end process stimulus;  
end bench;
```

E.2 Lift Control System Simulation Output

```
"*****"
Simulation of the Lift System specified in VHDL
"*****"
```

INITIAL VALUES ARE:

```
EMER_BUTTON = (L1_Value => FALSE, L2_Value => FALSE)
SUM_BUTTON  = (Floor => 0, Dir => NIL)
DEST_BUTTON = (Floor => 0, Lift => 0)

LIGHT_ON    = (Sum_Fl => 0, Sum_Dir => NIL,
               Dest_Fl => 0, Dest_Lift => 0)
LIGHT_OFF   = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP,
               Sum_Alt_Fl => 1, Sum_Alt_Dir => UP,
               Dest_Pri_Fl => 1, Dest_Pri_Lift => 1,
               Dest_Alt_Fl => 1, Dest_Alt_Lift => 2)
EMER_SIGNAL = (L1_Value => FALSE, L2_Value => FALSE)
MOTOR_IND   = (L1_Ind => IDLE, L2_Ind => IDLE)
LIFT_STATUS = (L1_Dir => UP, L1_Loc => 1,
               L2_Dir => UP, L2_Loc => 1)
```

SIMULATION VALUES ARE:

```
*****
TEST CASE 1: Two Lifts Moving with no Error Conditions.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 3 and 4.
*****
```

```
*****
Test 1: 2 UP SUMMONS REQUEST.
*****
```

```
0 SEC
SUM_BUTTON    = (Floor => 2, Dir => UP)

LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => UP)

MOTOR_IND     = (L1_Ind => RUN)

LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 0)
```

```
*****
Test 1: 3 DOWN SUMMONS REQUEST.
*****
```

```
1 SEC
SUM_BUTTON    = (Floor => 3, Dir => DOWN)
```

```

LIGHT_ON      = (Sum_Fl => 3, Sum_Dir => DOWN)

MOTOR_IND     = (L2_Ind => RUN)

LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 0)

2 SEC
10 SEC
LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 2)

MOTOR_IND     = (L1_Ind => STOP)

LIGHT_OFF     = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF     = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

MOTOR_IND     = (L1_Ind => IDLE)

*****
Test 1:  DESTINATION FLOOR 3 ON LIFT 1.
*****

11 SEC
DEST_BUTTON   = (Floor => 3, Lift => 1)

LIGHT_ON      = (Dest_Fl => 3, Dest_Lift => 1)

LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 2)

LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 0)

MOTOR_IND     = (L1_Ind => RUN)

LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 0)

12 SEC
21 SEC
LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 3)

LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 3)

LIFT_STATUS   = (L2_Dir => DOWN, L2_Loc => 3)

LIFT_STATUS   = (L1_Dir => DOWN, L1_Loc => 3)

MOTOR_IND     = (L2_Ind => STOP)

MOTOR_IND     = (L1_Ind => STOP)

LIGHT_OFF     = (Dest_Alt_Fl => 3, Dest_Alt_Lift => 2)

LIGHT_OFF     = (Sum_Alt_Fl => 3, Sum_Alt_Dir => DOWN)

LIGHT_OFF     = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)

```



```

LIGHT_OFF      = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)

MOTOR_IND      = (L2_Ind => IDLE)

MOTOR_IND      = (L1_Ind => IDLE)

*****
Test 1:  DESTINATION FLOOR 1 ON LIFT 2.
*****

22 SEC
DEST_BUTTON    = (Floor => 1, Lift => 2)

LIGHT_ON       = (Dest_Fl => 1, Dest_Lift => 2)

MOTOR_IND      = (L2_Ind => RUN)

LIFT_STATUS    = (L2_Dir => DOWN, L2_Loc => 0)

23 SEC
32 SEC
LIFT_STATUS    = (L2_Dir => DOWN, L2_Loc => 2)

LIFT_STATUS    = (L2_Dir => DOWN, L2_Loc => 0)

42 SEC
LIFT_STATUS    = (L2_Dir => DOWN, L2_Loc => 1)

LIFT_STATUS    = (L2_Dir => UP, L2_Loc => 1)

MOTOR_IND      = (L2_Ind => STOP)

LIGHT_OFF      = (Dest_Alt_Fl => 1, Dest_Alt_Lift => 2)

LIGHT_OFF      = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

MOTOR_IND      = (L2_Ind => IDLE)

*****
TEST CASE 2: Lift 1 Moving; Lift 2 in Emergency Condition.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 3, 5 and 6.
Note: The 2 UP SUMMONS occurs as the lift is going DOWN to
meet the DESTINATION request, so the lift does not stop.
*****

*****
Test 2:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.
*****

100 SEC
EMER_BUTTON    = (L2_Value => TRUE)

```

CURRENT_STATUS = (L2 => OUT_OF_SVC)

EMER_SIGNAL = (L2_Value => TRUE)

MOTOR_IND = (L2_Ind => STOP)

Test 2: 3 DOWN SUMMONS REQUEST.

101 SEC

SUM_BUTTON = (Floor => 3, Dir => DOWN)

LIGHT_ON = (Sum_Fl => 3, Sum_Dir => DOWN)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)

102 SEC

Test 2: DESTINATION FLOOR 1 ON LIFT 1

DEST_BUTTON = (Floor => 1, Lift => 1)

LIGHT_ON = (Dest_Fl => 1, Dest_Lift => 1)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

103 SEC

Test 2: 2 UP SUMMONS REQUEST.

SUM_BUTTON = (Floor => 2, Dir => UP)

LIGHT_ON = (Sum_Fl => 2, Sum_Dir => UP)

104 SEC

Test 2: DESTINATION FLOOR 1 ON LIFT 2.

105 SEC

DEST_BUTTON = (Floor => 1, Lift => 2)

LIGHT_ON = (Dest_Fl => 1, Dest_Lift => 2)

```

106 SEC
112 SEC
    LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 2)

    LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 0)

122 SEC
    LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 1)

    LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 1)

    MOTOR_IND        = (L1_Ind => STOP)

    LIGHT_OFF        = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

    LIGHT_OFF        = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

    MOTOR_IND        = (L1_Ind => RUN)

    LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 0)

132 SEC
    LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 2)

    MOTOR_IND        = (L1_Ind => STOP)

    LIGHT_OFF        = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

    LIGHT_OFF        = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

```

```

*****
TEST CASE 3: Both Lift 1 and Lift 2 in Emergency Conditions.
This test demonstrates design compliance with Problem
Constraint 6 and a design decision to disregard Summons
Requests when both Lifts are Out of Service.
*****

```

```

*****
Test 3: EMERGENCY BUTTONS ON BOTH LIFT 1 AND LIFT 2 PRESSED.
*****

```

```

200 SEC
    EMER_BUTTON      = (L1_Value => TRUE)

    CURRENT_STATUS    = (L1 => OUT_OF_SVC)

    EMER_SIGNAL      = (L1_Value => TRUE)

```

```

*****
Test 3: 3 DOWN SUMMONS REQUEST.
Test 3: Note that no lights are illuminated.
*****

```

201 SEC

SUM_BUTTON = (Floor => 3, Dir => DOWN)

202 SEC

TEST CASE 4: Lift 1 Moving; Lift 2 in Emergency Condition.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 3, 5, and 6. Note that the 2 UP
SUMMONS request occurs while going UP from floor 1, and it
is handled enroute.

Test 4: LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.

300 SEC

EMER_BUTTON = (L1_Value => FALSE)

LIGHT_OFF = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

EMER_SIGNAL = (L1_Value => FALSE)

Test 4: 2 DOWN SUMMONS REQUEST.

301 SEC

SUM_BUTTON = (Floor => 2, Dir => DOWN)

LIGHT_ON = (Sum_Fl => 2, Sum_Dir => DOWN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 2)

302 SEC

Test 4: DESTINATION FLOOR 1 ON LIFT 1.

DEST_BUTTON = (Floor => 1, Lift => 1)

LIGHT_ON = (Dest_Fl => 1, Dest_Lift => 1)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

303 SEC

```
*****
Test 4: 3 DOWN SUMMONS REQUEST.
*****
```

```
SUM_BUTTON      = (Floor => 3, Dir => DOWN)
```

```
LIGHT_ON        = (Sum_Fl => 3, Sum_Dir => DOWN)
```

```
304 SEC
```

```
*****
Test 4: 2 UP SUMMONS REQUEST.
*****
```

```
305 SEC
```

```
SUM_BUTTON      = (Floor => 2, Dir => UP)
```

```
LIGHT_ON        = (Sum_Fl => 2, Sum_Dir => UP)
```

```
306 SEC
```

```
312 SEC
```

```
LIFT_STATUS     = (L1_Dir => DOWN, L1_Loc => 1)
```

```
LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 1)
```

```
MOTOR_IND       = (L1_Ind => STOP)
```

```
LIGHT_OFF       = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)
```

```
LIGHT_OFF       = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)
```

```
MOTOR_IND       = (L1_Ind => RUN)
```

```
LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 0)
```

```
322 SEC
```

```
LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 2)
```

```
MOTOR_IND       = (L1_Ind => STOP)
```

```
LIGHT_OFF       = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)
```

```
LIGHT_OFF       = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)
```

```
MOTOR_IND       = (L1_Ind => RUN)
```

```
LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 0)
```

```
332 SEC
```

```
LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 3)
```

```
LIFT_STATUS     = (L1_Dir => DOWN, L1_Loc => 3)
```

```
MOTOR_IND       = (L1_Ind => STOP)
```

```
LIGHT_OFF       = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)
```

```

LIGHT_OFF      = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)

MOTOR_IND      = (L1_Ind => RUN)

LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 0)

```

```

*****
Test 4:  DESTINATION FLOOR 1 ON LIFT 1.
*****

```

```

333 SEC
  DEST_BUTTON    = (Floor => 1, Lift => 1)

  LIGHT_ON       = (Dest_Fl => 1, Dest_Lift => 1)

334 SEC
342 SEC
  LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 2)

  MOTOR_IND      = (L1_Ind => STOP)

  LIGHT_OFF      = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

  LIGHT_OFF      = (Sum_Pri_Fl => 2, Sum_Pri_Dir => DOWN)

  MOTOR_IND      = (L1_Ind => RUN)

  LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 0)

352 SEC
  LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 1)

  LIFT_STATUS    = (L1_Dir => UP, L1_Loc => 1)

  MOTOR_IND      = (L1_Ind => STOP)

  LIGHT_OFF      = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

  LIGHT_OFF      = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

  MOTOR_IND      = (L1_Ind => IDLE)

```

```

*****
TEST CASE 5: Both Lift 1 and Lift 2 in Service.
This test demonstrates design compliance with Problem
Constraints 1, 2, and 3. Note that both requests given are
for the floor where the lifts are currently located and no
action is taken except illuminating and extinguishing lights
*****

```

```

400 SEC
  EMER_BUTTON    = (L2_Value => FALSE)

```

```

LIGHT_OFF      = (Dest_Alt_Fl => 1, Dest_ALt_Lift => 2)

LIGHT_OFF      = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

EMER_SIGNAL    = (L2_Value => FALSE)

MOTOR_IND      = (L2_Ind => IDLE)

```

```

*****
Test 5:  1 UP SUMMONS REQUEST.
*****

```

```

401 SEC
SUM_BUTTON     = (Floor => 1, Dir => UP)

LIGHT_ON       = (Sum_Fl => 1, Sum_Dir => UP)

MOTOR_IND      = (L1_Ind => STOP)

LIGHT_OFF      = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF      = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

```

402 SEC

```

*****
Test 5:  DESTINATION FLOOR 1 ON LIFT 2.
*****

```

```

DEST_BUTTON    = (Floor => 1, Lift => 2)

LIGHT_ON       = (Dest_Fl => 1, Dest_Lift => 2)

MOTOR_IND      = (L2_Ind => STOP)

LIGHT_OFF      = (Dest_Alt_Fl => 1, Dest_ALt_Lift => 2)

LIGHT_OFF      = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

LIGHT_OFF      = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF      = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

```

403 SEC

```

*****
TEST CASE 6: Lift 1 Running; Lift 2 in Emergency Conditions.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 2(b), 3, 4, and 6. Note the waiting
time to handle both UP and DOWN requests on floor 2.
*****

```

```

*****
Test 6:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.

```

500 SEC

EMER_BUTTON = (L2_Value => TRUE)

CURRENT_STATUS = (L2 => OUT_OF_SVC)

LIGHT_OFF = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

EMER_SIGNAL = (L2_Value => TRUE)

Test 6: 1 UP SUMMONS REQUEST.

501 SEC

SUM_BUTTON = (Floor => 1, Dir => UP)

LIGHT_ON = (Sum_Fl => 1, Sum_Dir => UP)

502 SEC

Test 6: DESTINATION FLOOR 3 ON LIFT 1.

DEST_BUTTON = (Floor => 3, Lift => 1)

LIGHT_ON = (Dest_Fl => 3, Dest_Lift => 1)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 0)

503 SEC

Test 6: 2 DOWN SUMMONS REQUEST.

SUM_BUTTON = (Floor => 2, Dir => DOWN)

LIGHT_ON = (Sum_Fl => 2, Sum_Dir => DOWN)

504 SEC

512 SEC

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 2)

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 0)

Test 6: 2 UP SUMMONS REQUEST.

518 SEC

SUM_BUTTON = (Floor => 2, Dir => UP)

LIGHT_ON = (Sum_Fl => 2, Sum_Dir => UP)

519 SEC

522 SEC

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 3)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 3)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

532 SEC

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 2)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 2, Sum_Pri_Dir => DOWN)

Test 6: DESTINATION FLOOR 1 ON LIFT 1.

533 SEC

DEST_BUTTON = (Floor => 1, Lift => 1)

LIGHT_ON = (Dest_Fl => 1, Dest_Lift => 1)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

534 SEC

543 SEC

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 1)

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 1)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

```

LIGHT_OFF      = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

MOTOR_IND      = (L1_Ind => RUN)

LIFT_STATUS    = (L1_Dir => UP, L1_Loc => 0)

553 SEC
LIFT_STATUS    = (L1_Dir => UP, L1_Loc => 2)

MOTOR_IND      = (L1_Ind => STOP)

LIGHT_OFF      = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF      = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 2)

MOTOR_IND      = (L1_Ind => IDLE)

```

```

*****
TEST CASE 7: Lift 1 Running; Lift 2 in Emergency Conditions.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 2(b), 3, 4, and 6. Note the waiting
time to handle both UP and DOWN requests on floor 2.
*****

```

```

*****
Test 7:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.
*****

```

```

*****
Test 7:  1 UP SUMMONS REQUEST.
*****

```

```

601 SEC
SUM_BUTTON     = (Floor => 1, Dir => UP)

LIGHT_ON       = (Sum_Fl => 1, Sum_Dir => UP)

MOTOR_IND      = (L1_Ind => RUN)

LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 0)

602 SEC
611 SEC
LIFT_STATUS    = (L1_Dir => DOWN, L1_Loc => 1)

LIFT_STATUS    = (L1_Dir => UP, L1_Loc => 1)

MOTOR_IND      = (L1_Ind => STOP)

LIGHT_OFF      = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

```

```

LIGHT_OFF      = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

MOTOR_IND      = (L1_Ind => IDLE)

*****
Test 7:  DESTINATION FLOOR 3 ON LIFT 1.
*****

612 SEC
  DEST_BUTTON   = (Floor => 3, Lift => 1)

  LIGHT_ON      = (Dest_Fl => 3, Dest_Lift => 1)

  MOTOR_IND     = (L1_Ind => RUN)

  LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 0)

613 SEC

*****
Test 7:  2 DOWN SUMMONS REQUEST.
*****

  SUM_BUTTON    = (Floor => 2, Dir => DOWN)

  LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => DOWN)

614 SEC

*****
Test 7:  2 UP SUMMONS REQUEST.
*****

615 SEC
  SUM_BUTTON    = (Floor => 2, Dir => UP)

  LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => UP)

616 SEC
622 SEC
  LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 2)

  MOTOR_IND     = (L1_Ind => STOP)

  LIGHT_OFF     = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

  LIGHT_OFF     = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

  MOTOR_IND     = (L1_Ind => RUN)

  LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 0)

*****
Test 7:  DESTINATION FLOOR 1 ON LIFT 1.
*****

```

623 SEC

DEST_BUTTON = (Floor => 1, Lift => 1)

LIGHT_ON = (Dest_Fl => 1, Dest_Lift => 1)

624 SEC

632 SEC

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 3)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 3)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

642 SEC

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 2)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 2, Sum_Pri_Dir => DOWN)

MOTOR_IND = (L1_Ind => RUN)

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 0)

652 SEC

LIFT_STATUS = (L1_Dir => DOWN, L1_Loc => 1)

LIFT_STATUS = (L1_Dir => UP, L1_Loc => 1)

MOTOR_IND = (L1_Ind => STOP)

LIGHT_OFF = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

MOTOR_IND = (L1_Ind => IDLE)

TEST CASE 8: Lift 1 Running; Lift 2 in Emergency Conditions.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 2(b), 3, 4, and 6. Note the waiting
time to handle both UP and DOWN requests on floor 2.

```
*****
Test 8:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 PRESSED.
*****
```

```
*****
Test 8:  1 UP SUMMONS REQUEST.
*****
```

```
701 SEC
SUM_BUTTON      = (Floor => 1, Dir => UP)

LIGHT_ON        = (Sum_Fl => 1, Sum_Dir => UP)

MOTOR_IND       = (L1_Ind => STOP)

LIGHT_OFF       = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF       = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)
```

702 SEC

```
*****
Test 8:  DESTINATION FLOOR 2 ON LIFT 1.
*****
```

```
712 SEC
DEST_BUTTON     = (Floor => 2, Lift => 1)

LIGHT_ON        = (Dest_Fl => 2, Dest_Lift => 1)

MOTOR_IND       = (L1_Ind => RUN)

LIFT_STATUS     = (L1_Dir => UP, L1_Loc => 0)
```

713 SEC

```
*****
Test 8:  2 DOWN SUMMONS REQUEST.
*****
```

```
SUM_BUTTON      = (Floor => 2, Dir => DOWN)

LIGHT_ON        = (Sum_Fl => 2, Sum_Dir => DOWN)
```

714 SEC

```
*****
Test 8:  2 UP SUMMONS REQUEST.
*****
```

```
715 SEC
SUM_BUTTON      = (Floor => 2, Dir => UP)
```

```

LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => UP)

716 SEC
722 SEC
LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 2)

MOTOR_IND     = (L1_Ind => STOP)

LIGHT_OFF     = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF     = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

LIFT_STATUS   = (L1_Dir => DOWN, L1_Loc => 2)

MOTOR_IND     = (L1_Ind => IDLE)

```

```

*****
Test 8:  DESTINATION FLOOR 1 ON LIFT 1.
*****

```

```

723 SEC
DEST_BUTTON   = (Floor => 1, Lift => 1)

LIGHT_ON      = (Dest_Fl => 1, Dest_Lift => 1)

MOTOR_IND     = (L1_Ind => RUN)

LIFT_STATUS   = (L1_Dir => DOWN, L1_Loc => 0)

724 SEC
733 SEC
LIFT_STATUS   = (L1_Dir => DOWN, L1_Loc => 1)

LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 1)

MOTOR_IND     = (L1_Ind => STOP)

LIGHT_OFF     = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

LIGHT_OFF     = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

MOTOR_IND     = (L1_Ind => IDLE)

```

```

*****
TEST CASE 9: Both Lift 1 and Lift 2 Running.
This test demonstrates design compliance with Problem
Constraints 1, 2(a), 2(b), 3, 4, and 6.  Note the waiting
time to handle both UP and DOWN requests on floor 2.
*****

```

```

*****
Test 9:  LIFT 1 RUNNING; EMERGENCY BUTTON ON LIFT 2 CLEARED.
*****

```

```

800 SEC
  EMER_BUTTON      = (L2_Value => FALSE)

  LIGHT_OFF        = (Dest_Alt_Fl => 1, Dest_Alt_Lift => 2)

  LIGHT_OFF        = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

  EMER_SIGNAL      = (L2_Value => FALSE)

*****
Test 9:  1 UP SUMMONS REQUEST.
*****

801 SEC
  SUM_BUTTON       = (Floor => 1, Dir => UP)

  LIGHT_ON         = (Sum_Fl => 1, Sum_Dir => UP)

  MOTOR_IND       = (L1_Ind => STOP)

  LIGHT_OFF        = (Dest_Alt_Fl => 1, Dest_Alt_Lift => 2)

  LIGHT_OFF        = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

  LIGHT_OFF        = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)

  LIGHT_OFF        = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)

802 SEC

*****
Test 9:  DESTINATION FLOOR 3 ON LIFT 1.
*****

812 SEC
  DEST_BUTTON      = (Floor => 3, Lift => 1)

  LIGHT_ON         = (Dest_Fl => 3, Dest_Lift => 1)

  MOTOR_IND       = (L1_Ind => RUN)

  LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 0)

  LIGHT_OFF        = (Dest_Alt_Fl => 1, Dest_Alt_Lift => 2)

  LIGHT_OFF        = (Sum_Alt_Fl => 1, Sum_Alt_Dir => UP)

813 SEC

*****
Test 9:  2 DOWN SUMMONS REQUEST.
*****

  SUM_BUTTON       = (Floor => 2, Dir => DOWN)

```

```

LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => DOWN)

MOTOR_IND     = (L2_Ind => RUN)

LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 0)

814 SEC

*****
Test 9:  2 UP SUMMONS REQUEST.
*****

815 SEC
SUM_BUTTON    = (Floor => 2, Dir => UP)

LIGHT_ON      = (Sum_Fl => 2, Sum_Dir => UP)

816 SEC
822 SEC
LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 2)

MOTOR_IND     = (L1_Ind => STOP)

LIGHT_OFF     = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)

LIGHT_OFF     = (Sum_Pri_Fl => 2, Sum_Pri_Dir => UP)

MOTOR_IND     = (L1_Ind => RUN)

LIFT_STATUS   = (L1_Dir => UP, L1_Loc => 0)

*****
Test 9:  DESTINATION FLOOR 1 ON LIFT 1.
*****

823 SEC
DEST_BUTTON   = (Floor => 1, Lift => 1)

LIGHT_ON      = (Dest_Fl => 1, Dest_Lift => 1)

LIFT_STATUS   = (L2_Dir => UP, L2_Loc => 2)

LIFT_STATUS   = (L2_Dir => DOWN, L2_Loc => 2)

MOTOR_IND     = (L2_Ind => STOP)

LIGHT_OFF     = (Dest_Alt_Fl => 2, Dest_Alt_Lift => 2)

LIGHT_OFF     = (Sum_Alt_Fl => 2, Sum_Alt_Dir => DOWN)

MOTOR_IND     = (L2_Ind => IDLE)

824 SEC
832 SEC

```



```

LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 3)
LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 3)
MOTOR_IND        = (L1_Ind => STOP)
LIGHT_OFF        = (Dest_Pri_Fl => 3, Dest_Pri_Lift => 1)
LIGHT_OFF        = (Sum_Pri_Fl => 3, Sum_Pri_Dir => DOWN)
MOTOR_IND        = (L1_Ind => RUN)
LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 0)

842 SEC
LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 2)
MOTOR_IND        = (L1_Ind => STOP)
LIGHT_OFF        = (Dest_Pri_Fl => 2, Dest_Pri_Lift => 1)
LIGHT_OFF        = (Sum_Pri_Fl => 2, Sum_Pri_Dir => DOWN)
MOTOR_IND        = (L1_Ind => RUN)
LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 0)

852 SEC
LIFT_STATUS      = (L1_Dir => DOWN, L1_Loc => 1)
LIFT_STATUS      = (L1_Dir => UP, L1_Loc => 1)
MOTOR_IND        = (L1_Ind => STOP)
LIGHT_OFF        = (Dest_Pri_Fl => 1, Dest_Pri_Lift => 1)
LIGHT_OFF        = (Sum_Pri_Fl => 1, Sum_Pri_Dir => UP)
MOTOR_IND        = (L1_Ind => IDLE)

900 SEC
Assertion ERROR at 900 SEC in design unit BENCH from process /SYSTEM/TESTBENCH/STIMULUS:
  "SIMULATION TESTING COMPLETE."

```

Appendix F. *Objects and Operations of the Essential Data Model for IDEF₀*

Diagrams

This report was written by Douglass (16) and provides information about object classes and operations from the essential data model representation of IDEF₀ (SADT) diagrams. The information was extracted from the source code authored by Captain Terry Kitchen. The first part of the report gives a simple summary of each of the object classes, and lists the record fields along with the operations that the class provides. The second section provides a more detailed description of each of the operations.

F.1 ACTIVITY-CLASS

PURPOSE: This package exports the activity object class. An activity is a record type consisting of several components. Those component types are also exported to the client.

F.1.1 Record Fields

Name:	<code>Activity-Name-Type := Null-Activity-Name;</code>
Number:	<code>Activity-Number-Type := Null-Activity-Number;</code>
Description:	<code>Environment-Types.Text-Buffer-Package.Manager-Type;</code>
Children:	<code>Environment-Types.Data-Buffer-Package.Manager-Type;</code>
Reference-Type:	<code>Environment-Types.Reference-Type := Environment-Types.Null-Reference-Type;</code>
Reference:	<code>Environment-Types.Text-Buffer-Package.Manager-Type;</code>
Version:	<code>Activity-Version-Type := Null-Activity-Version-Number;</code>
Version-Changes:	<code>Environment-Types.Text-Buffer-Package.Manager-Type;</code>
Date:	<code>Environment-Types.Date-Type := Environment-Types.Null-Date;</code>
Author:	<code>Environment-Types.Author-Name-Type := Environment-Types.Null-Author-Name;</code>

F.1.2 Operations

- Clear-Activity-Manager
- Activity-Is-Child
- Create-Activity
- Kill-Activity
- Activity-Exists
- Add-Activity-Child
- Delete-Activity-Child
- Delete-Activity-Children-List
- Change-Activity-Name-In-Hierarchy

- Set-Activity-Number
- Set-Activity-Description
- Set-Activity-Reference-Type
- Set-Activity-Reference
- Set-Activity-Version
- Set-Activity-Version-Comments
- Set-Activity-Date
- Set-Activity-Author
- Value-Of-Activity
- Activity-Is-Parent
- Activity-Is-Grandfather
- Number-Of-Activities
- Reset-Activity-Iterator
- Value-Of-Activity-At-Iterator
- Advance-Iterator-To-Next-Activity
- Activity-Iterator-Done

F.2 DATA-ELEMENT-CLASS

PURPOSE: This package exports the object class of "Data-Element". "Data-Element" is a record type consisting of several components. Those component types are also exported to the client.

F.2.1 Record Fields

```

Name:      Data-Element-Name-Type:= Null-Data-Element-Name;
Data-Type: Data-Element-Data-Type:= Null-Data-Element-Data-Type;
Minimum:   Data-Element-Value-Type:= Null-Data-Element-Value;
Maximum:   Data-Element-Value-Type:= Null-Data-Element-Value;
Data-Range: Data-Element-Value-Type:= Null-Data-Element-Value;
Values:    Environment-Types.Data-Buffer-Package.Manager-Type;
Description: Environment-Types.Text-Buffer-Package.Manager-Type;
Reference:  Environment-Types.Text-Buffer-Package.Manager-Type;
Reference-Type: Environment-Types.Reference-Type:=
              Environment-Types.Null-Reference-Type;
Version: Data-Element-Version-Type:= Null-Data-Element-Version-Number;
Version-Changes: Environment-Types.Text-Buffer-Package.Manager-Type;
Date:      Environment-Types.Date-Type:= Environment-Types.Null-Date;
Author:    Environment-Types.Author-Name-Type:=
              Environment-Types.Null-Author-Name;

```

F.2.2 Operations

- Clear-Data-Element-Manager
- Create-Data-Element
- Kill-Data-Element
- Data-Element-Exists
- Set-Data-Element-Name
- Set-Data-Element-Data-Type
- Set-Data-Element-Minimum
- Set-Data-Element-Maximum
- Set-Data-Element-Data-Range
- Set-Data-Element-Description
- Set-Data-Element-Values
- Set-Data-Element-Reference-Type
- Set-Data-Element-Reference
- Set-Data-Element-Version
- Set-Data-Element-Version-Comments
- Set-Data-Element-Date
- Set-Data-Element-Author
- Value-Of-Data-Element
- Number-Of-Data-Elements
- Reset-Data-Element-Iterator
- Value-Of-Data-Element-At-Iterator
- Advance-Iterator-To-Next-Data-Element
- Data-Element-Iterator-Done

F.3 ICOM-RELATION-CLASS

PURPOSE: This package exports the object class of "ICOM-Relation- Class" which is a record type consisting of four components.

F.3.1 Record Fields

```
Pair-Id:      natural:= 0;
Activity:     Activity-Class.Activity-Name-Type:=
              Activity-Class.Null-Activity-Name;
Data-Element: Data-Element-Class.Data-Element-Name-Type:=
              Data-Element-Class.Null-Data-Element-Name;
Relationship: character:= ' ';
```

F.3.2 Operations

- Clear-ICOM-Relation-Manager
- Create-ICOM-Relation-Tuple
- Kill-ICOM-Relation-Tuple
- Kill-ICOM-Relation-Tuples-With-Activity-Name
- Kill-ICOM-Relation-Tuples-With-Data-Element
- Kill-ICOM-Relation-Tuples-With-Pair-Id
- ICOM-Relation-Change-Activity-Name
- ICOM-Relation-Tuple-Exists
- ICOM-Relation-Tuple-With-Activity-Name-Exists
- ICOM-Relation-Tuple-With-Data-Element-Exists
- Value-Of-Next-ICOM-Pair-Id
- Value-Of-ICOM-Counts
- Value-Of-ICOM-Pair-Id
- Value-Of-ICOM-Relation-Tuple
- Number-Of-ICOM-Relation-Tuples
- Reset-ICOM-Relation-Tuple-Iterator
- Value-Of-ICOM-Relation-Tuple-At-Iterator
- Advance-Iterator-To-Next-ICOM-Relation-Tuple
- ICOM-Relation-Tuple-Iterator-Done

F.4 CALLS-RELATION-CLASS

PURPOSE: This package exports the object class of "Calls-Relation- Class" which is a record type consisting of two components. These components are an activity name and the historical activity tuple which it is related to.

F.4.1 Record Fields

```
Activity:      Activity-Class.Activity-Name-Type:=  
               Activity-Class.Null-Activity-Name;  
History-Tuple: Historical-Activity-Class.  
               Historical-Activity-Record-Type;
```

F.4.2 Operations

- Clear-Calls-Relation-Manager
- Create-Calls-Relation-Tuple
- Kill-Calls-Relation-Tuple
- Kill-Calls-Relation-Tuples-With-Activity-Name
- Calls-Relation-Tuple-Exists
- Calls-Relation-Tuple-With-Activity-Name-Exists
- Value-Of-Calls-Relation-Tuple
- Number-Of-Calls-Relation-Tuples
- Reset-Calls-Relation-Tuple-Iterator
- Value-Of-Calls-Relation-Tuple-At-Iterator
- Advance-Iterator-To-Next-Calls-Relation-Tuple
- Calls-Relation-Tuple-Iterator-Done

F.5 CONSISTS-OF-RELATION-CLASS

PURPOSE: This package exports the object class of "Consists-Of". "Consists-Of" is a record type with three components. These 3 components together uniquely identify a part of a data element decomposition. For example, if data element "m" is a *composite data element* that consists of data elements "x", "y", and "z" then three objects - [1,m,x], [1,m,y], [1,m,z] would be required and stored in a manager (see the "consists-of-manager-package").

F.5.1 Record Fields

```
Consists-Of-Id: natural;  
Parent:      Data-Element-Class.Data-Element-Name-Type;  
Child:      Data-Element-Class.Data-Element-Name-Type;
```

F.5.2 Operations

- Clear-Consists-Of-Relation-Manager
- Create-Consists-Of-Relation-Tuple
- Kill-Consists-Of-Relation-Tuple
- Kill-Consists-Of-Relation-Tuples-With-Consists-Of-Id
- Consists-Of-Relation-Tuple-Exists
- Value-Of-Next-Consists-Of-Id
- Value-Of-Consists-Of-Id (given a child list rtn a number)
- Consists-Of-Relation-Tuple-With-Parent-DE-Exists
- Consists-Of-Relation-Tuple-With-Child-DE-Exists
- Consists-Of-Relation-Tuple-With-Parent-Child-DE-Exists

- Value-Of-Consists-Of-Relation-Tuple
- Number-Of-Consists-Of-Relation-Tuples
- Reset-Consists-Of-Relation-Tuple-Iterator
- Value-Of-Consists-Of-Relation-Tuple-At-Iterator
- Advance-Iterator-To-Next-Consists-Of-Relation-Tuple
- Consists-Of-Relation-Tuple-Iterator-Done

F.6 HISTORICAL-ACTIVITY-CLASS

PURPOSE: This package exports the object class of "Historical- Activity" which is a record type consisting of two components. These components are a project name which can be any name (including the current project) plus an activity number from the project. No cross-checking is done at this level to determine if the project names are valid or if the activity numbers correspond to that project.

F.6.1 Record Fields

```
Project:      Environment-Types.Project-Name-Type:=
              Environment-Types.Null-Project-Name;
Activity-Number: Activity-Class.Activity-Number-Type:=
              Activity-Class.Null-Activity-Number;
```

F.6.2 Operations

- Clear-Historical-Activity-Manager
- Create-Historical-Activity
- Kill-Historical-Activity
- Historical-Activity-Exists
- Value-Of-Historical-Activity
- Number-Of-Historical-Activities
- Reset-Historical-Activity-Iterator
- Value-Of-Historical-Activity-At-Iterator
- Advance-Iterator-To-Next-Historical-Activity
- Historical-Activity-Iterator-Done

F.7 Detailed Operation Descriptions

F.7.1 Activity

- CLEAR-ACTIVITY-MANAGER

DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".

- **ACTIVITY-EXISTS**

DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if an activity exists in the manager. Second, if the activity does exist, the pointer value is updated to point to that activity. Note that the local variable "local-pointer" is necessary because the instantiated generic package, Activity-Manager-Package expects pointers of type "Iterator-Type", thus a type conversion using "local-pointer" is required.

- **ACTIVITY-IS-CHILD**

DESCRIPTION: This procedure simply determines if a given activity name appears as a child anywhere. If so a boolean flag is passed back as True. The Parent Name is also returned. The procedure stops when an activity that has a matching child name is found since an activity can only be in 1 child list (i.e., only 1 parent is allowed).

- **CREATE-ACTIVITY**

DESCRIPTION: This procedure creates a new activity within the the activity manager. If an activity with the same name does not already exist, then it simply creates a new activity and assigns the name. "Activity-Pointer" is used to pass back to the calling procedure the address of the new activity. There are 3 conditions that must be met before an activity is added: 1) the name cannot be blanks. 2) another activity with the same name can't exist. 3) another activity which has a child in its child list with the same name can't already exist.

- **CHANGE-ACTIVITY-NAME-IN-HIERARCHY**

DESCRIPTION: This procedure first checks to make sure that the new activity name does not already appear somewhere in the hierarchy. If the new name does not appear anywhere, then the procedure simply iterates through the manager and wherever it finds an old name, that name is replace with the new name, including the child lists!!

- **KILL-ACTIVITY**

DESCRIPTION: This procedure simply removes the activity from the manager at the location pointed to by the "Activity-Pointer". The key feature of this procedure is that the activity pointer is returned with a null value which prevents potential user abuse.

- **ADD-ACTIVITY-CHILD**

DESCRIPTION: A child is added to an activity if both the parent and child activities already exist AND the child activity name does NOT appear as a child for any other activity (that would give it two parents which is not allowed).

- **DELETE-ACTIVITY-CHILD**

DESCRIPTION: A child can be deleted from the children list of an activity with the only check being that the parent activity must obviously exist. It is NOT necessary for the activity represented by the child-activity-name to exist. This allows an activity to exist without being part of the hierarchy. If at some future point such flexibility is no longer desired, then this module will require modifications.

- **DELETE-ACTIVITY-CHILDREN-LIST**

DESCRIPTION: This procedure can be useful when an entire decomposition of an activity needs to be deleted but not the activity itself. The procedure checks to make sure the parent activity exists before attempting to delete the children.

- **SET-ACTIVITY-NUMBER**

DESCRIPTION: This procedure sets the value of the activity number to the value provided in the parameter "Act-Number".

- **SET-ACTIVITY-DESCRIPTION**

DESCRIPTION: This procedure sets the value of the activity description to the value provided in the parameter "Act-Descript".

- **SET-ACTIVITY-REFERENCE-TYPE**
DESCRIPTION: This procedure sets the value of the reference-type attribute which is associated with the reference attribute.
- **SET-ACTIVITY-REFERENCE**
DESCRIPTION: This procedure sets the value of the activity reference which is a multi-line text field. Comments".
- **SET-ACTIVITY-VERSION**
DESCRIPTION: This procedure sets the value of the activity version to the value provided in the parameter "Act-Version". raise Activity-Iterator-Error; end Set-Activity-Version;
- **SET-ACTIVITY-VERSION-COMMENTS**
DESCRIPTION: This procedure sets the value of the activity version comments to the value provided in the parameter "Act-Version- Comments".
- **SET-ACTIVITY-DATE**
DESCRIPTION: This procedure sets the value of the activity date to the value provided in the parameter "Act-Date".
- **SET-ACTIVITY-AUTHOR**
DESCRIPTION: This procedure sets the value of the activity author to the value provided in the parameter "Act-Author".
- **VALUE-OF-ACTIVITY**
DESCRIPTION: There are two functions that retrieve activities. "Value-Of-Activity-At-Iterator" is for use with the iterator. The purpose of this function is to retrieve an entire record at the location given by the pointer. Its primary purpose is for use in getting quick access to all the activity information.
- **ACTIVITY-IS-PARENT**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one child in the children part of an activity record.
- **ACTIVITY-IS-GRANDFATHER**
DESCRIPTION: This function returns a boolean value indicating if the activity name passed as a parameter is, in fact, at least a grandfather in the hierarchy. Use this function prior to a delete being performed. For example, if a user wishes to delete an activity, the delete action should not be allowed if more than one lower level exists. Therefore, it's much the Unix OS where you cannot delete a directory that contains a directory. This function allows the client to check for more than 1 lower level prior to a delete - if True then the delete should not be performed (i.e., make the user delete the sub-directory first).
- **NUMBER-OF-ACTIVITIES**
DESCRIPTION: This function returns the number of activities in the activity manager. It returns 0 if it is null. Therefore, this function eliminates the need for an "Activity-Manager-Null?" function.
- **RESET-ACTIVITY-ITERATOR**
DESCRIPTION: This procedure sets the global activity iterator to point to the first activity in the manager.
- **VALUE-OF-ACTIVITY-AT-ITERATOR**
DESCRIPTION: This procedure retrieves the activity record at the position pointed to by the global variable "Activity-Iterator".

- **ADVANCE-ITERATOR-TO-NEXT-ACTIVITY**
DESCRIPTION: This procedure advances the global activity iterator to point to the next activity.
- **ACTIVITY-ITERATOR-DONE**
DESCRIPTION: This function returns a boolean value indicating if the global activity iterator is pointing to a null (done) position.

F.7.2 Data-Element

- **CLEAR-DATA-ELEMENT-MANAGER**
DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".
- **DATA-ELEMENT-EXISTS**
DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if a data element exists in the manager. Second, if the data element does exist the pointer value is updated to point to that data element. The local variable "local-pointer" is necessary because the instantiated package Data-Element-Manager-Package expects ptrs of type "Iterator-Type", thus a type conversion using "local-pointer" is required.
- **CREATE-DATA-ELEMENT**
DESCRIPTION: This procedure creates a new data element within the the manager. If a data element with the same name does not already exist, then it simply creates a new element and assigns the name. "Data-Element-Pointer" is used to pass back to the calling procedure the address of the new data element.
- **KILL-DATA-ELEMENT**
DESCRIPTION: This procedure simply removes a data element from the manager at the location pointed to by the "Data-Element-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse.
- **SET-DATA-ELEMENT-NAME**
DESCRIPTION: This procedure changes the name associated with a data element at the spot pointed to by the "Data-Element-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse. The tool (or perhaps the "expert" part) will have to make sure the name is changed elsewhere in the IDEF0 model in other relationships.
- **SET-DATA-ELEMENT-DATA-TYPE**
DESCRIPTION: This procedure sets the value of the data element data type to the value provided in the parameter "Data-Element-Data- Type".
- **SET-DATA-ELEMENT-MINIMUM**
DESCRIPTION: This procedure sets the value of the data element min value to the value provided in the parameter "Data-Element- Minimum".
- **SET-DATA-ELEMENT-MAXIMUM**
DESCRIPTION: This procedure sets the value of the data element min value to the value provided in the parameter "Data-Element- Maximum".
- **SET-DATA-ELEMENT-DATA-RANGE**
DESCRIPTION: This procedure sets the value of the data element data range to the value provided in the parameter "Data-Element- Range".
- **SET-DATA-ELEMENT-DESCRIPTION**
DESCRIPTION: This procedure sets the value of the data element description to the value provided in "Data-Element-Descript".

- **SET-DATA-ELEMENT-VALUES**
DESCRIPTION: This procedure sets the values attribute to a list of allowable values. This list is intended to be used when the min, max, and range attributes do not apply. In fact, this attribute should probably be a list of enumerated types. However, to save instantiating another generic, a string is used for a single value.
- **SET-DATA-ELEMENT-REFERENCE-TYPE**
DESCRIPTION: This procedure sets the reference-type attribute based on the multi-line reference attribute field. This field simply describes the type of reference in the reference attribute.
- **SET-DATA-ELEMENT-REFERENCE**
DESCRIPTION: This procedure sets the value of the data element reference to one or more lines of text.
- **SET-DATA-ELEMENT-VERSION**
DESCRIPTION: This procedure sets the value of the data element version to the value provided in "Data-Element-Version". -
- **SET-DATA-ELEMENT-VERSION-COMMENTS**
DESCRIPTION: This procedure sets the value of the data element version comments to the value provide in the parameter.
- **SET-DATA-ELEMENT-DATE**
DESCRIPTION: This procedure sets the value of the data element date to the value provided in the parameter "Data-Element-Date".
- **SET-DATA-ELEMENT-AUTHOR**
DESCRIPTION: This procedure sets the value of the data element author to the value passed via parameter in "Data-Element-Author".
- **VALUE-OF-DATA-ELEMENT**
DESCRIPTION: There are two functions that retrieve data elements. "Value-Of-Data-Element-At-Iterator" is used with the iterator. The purpose of this function is to retrieve an entire record at the location given by the pointer. Its primary purpose is for use in getting quick access to all the data element information.
- **NUMBER-OF-DATA-ELEMENTS**
DESCRIPTION: This function returns the number of data elements in the data element manager. A 0 is returned if empty. Thus, this function eliminates the need for an "Data-Element-Manager-Null?" function. One could argue that it should be included because it would be $O(1)$ vs $O(d)$; but note that if the manager is empty, this function is $O(1)$, so you only sacrifice when its not empty.
- **RESET-DATA-ELEMENT-ITERATOR**
DESCRIPTION: This procedure sets the global data element iterator to point to the first data element in the manager.
- **VALUE-OF-DATA-ELEMENT-AT-ITERATOR**
DESCRIPTION: This procedure retrieves the data element record at the position pointed to by the global (to this body) variable "Data-Element-Iterator".
- **ADVANCE-ITERATOR-TO-NEXT-DATA-ELEMENT**
DESCRIPTION: This procedure advances the global data element iterator to point to the next data element in the manager.
- **DATA-ELEMENT-ITERATOR-DONE**
DESCRIPTION: This function returns a boolean value indicating if the global data element iterator is pointing to a null (done) position.

F.7.3 ICOM-Relation

- **CLEAR-ICOM-RELATION-MANAGER**
DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".
- **ICOM-RELATION-TUPLE-EXISTS**
DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if an ICOM tuple exists in the mgr. Second, if the ICOM tuple exists, it also updates the pointer to the proper address. The local variable "local-pointer" is necessary because the instantiated package ICOM-Relation-Manager-Package expects ptrs of type "Iterator-Type" as parameters to its procedures and functions, thus a type conversion using "local-pointer" is required.
- **CREATE-ICOM-RELATION-TUPLE**
DESCRIPTION: This procedure creates a new ICOM tuple in the manager. If a ICOM tuple with the same key does not already exist, then it simply creates a new element.
- **KILL-ICOM-RELATION-TUPLE**
DESCRIPTION: This procedure simply removes a ICOM tuple in the mgr at the address of the "ICOM-Relation-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse.
- **KILL-ICOM-RELATION-TUPLES-WITH-ACTIVITY-NAME**
DESCRIPTION: This procedure removes all ICOM tuples that have a specific activity name. This, procedure could be used by a client when deleting a box from a diagram (i.e., when the box is removed, all its relationships must go with it).
- **KILL-ICOM-RELATION-TUPLES-WITH-DATA-ELEMENT**
DESCRIPTION: This procedure removes all ICOM tuples that have a specific data element name in the data element field. However, we must take note that there can be several instances of a data element being related to different activities in a IDEF0 model; therefore, we must include the activity also to insure we make no unintentional deletions of relationships. The most likely use for this procedure would occur when deleting from a diagram an arrow (data element) that had been connected to an activity (box).
- **KILL-ICOM-RELATION-TUPLES-WITH-PAIR-ID**
DESCRIPTION: This procedure removes all ICOM tuples that have a specific ICOM pair id in the pair id field. This procedure may, in fact, may be the easiest way for the drawing model to perform deletions relating to data elements. For example, when deleting a data element, use Value-Of-ICOM-Pair-Id to get the id number related to that specific data element, then execute this procedure and one or maybe two tuples will be deleted.
- **ICOM-RELATION-TUPLE-WITH-ACTIVITY-NAME-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one tuple with the given activity name. The function stops when the first tuple with a matching name is found.
- **ICOM-RELATION-TUPLE-WITH-DATA-ELEMENT-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one tuple with the given data element name. The function stops when the first tuple with a matching name is found.
- **ICOM-RELATION-CHANGE-ACTIVITY-NAME**
DESCRIPTION: This procedure changes the name of a given activity to a new name. Of course, the procedure must first check that the new-activity-name does not already appear in the manager. (i.e., you cannot change to a name that already exists) NOTE: It is the

responsibility of the client procedure to perform the necessary cross-checks with the other managers to insure the new name was not already present before the series of operations which include this one were queued up. This procedure only checks within itself for a preexisting name prior to changing all names.

- **VALUE-OF-NEXT-ICOM-PAIR-ID**

DESCRIPTION: This function simply returns a natural value indicates the next available id number for a SOURCE DESTINATION pair. It simply finds the highest number, then adds one.

- **VALUE-OF-ICOM-COUNTS**

DESCRIPTION: This procedure accepts an activity name and passes back via 4 parameters the number of inputs, outputs, controls and mechanisms this activity has. If the activity is not found, all zeroes are returned.

- **VALUE-OF-ICOM-PAIR-ID**

DESCRIPTION: Given that the drawing model has previously updated either the SOURCE or DESTINATION for a particular data element, and that SOURCE or DESTINATION is a box (activity), then let's say the other end of the data element is now being connected to a box. This relationship must be "paired" with the tuple already present in the manager; thus, we must find it and return its id so it can be used again to create a new tuple. The two tuples together would thus form a SOURCE - DESTINATION pair. Please note the following: 1) The pair-id in the input record is ignored. 2) If the tuple is not found the function returns a Zero value!

- **VALUE-OF-ICOM-RELATION-TUPLE**

DESCRIPTION: There are two functions that can retrieve a ICOM tuple. "Value-Of-ICOM-Relation-Tuple-At-Iterator", however, is for use only in conjunction with the other iterator functions and procedures. The purpose of this function is to retrieve the entire tuple at the location provided by the pointer. Note that because the entire tuple is the key, the function is not really necessary since you must supply the tuple to the "ICOM-Relation-Tuple-Exists" procedure before this function. Thus, a purist could claim this function to be "window dressing". However, it is always possible one or more non-key attributes may later be added; therefore, this function would be required given the implementation technique chosen by the author.

- **NUMBER-OF-ICOM-RELATION-TUPLES**

DESCRIPTION: This function returns the number tuples in the manager. A 0 is returned if empty. Thus, this function fills in for the "ICOM-Relation-Tuple-Manager-Null?" function. One could argue that it should be included because it would be $O(1)$ vs $O(i)$; but note that if the manager is empty, this function is $O(1)$, so you only sacrifice when its not empty.

- **RESET-ICOM-RELATION-TUPLE-ITERATOR**

DESCRIPTION: This procedure sets the global iterator to point to the first item in the manager.

- **VALUE-OF-ICOM-RELATION-TUPLE-AT-ITERATOR**

DESCRIPTION: This procedure retrieves the item in the manager at the position pointed to by the global (to this body) variable "ICOM-Relation-Iterator".

- **ADVANCE-ITERATOR-TO-NEXT-ICOM-RELATION-TUPLE**

DESCRIPTION: This procedure advances the global iterator to point to the next item in the manager.

- **ICOM-RELATION-TUPLE-ITERATOR-DONE**

DESCRIPTION: This function returns a boolean value indicating if the global iterator is pointing to a null (done) position.

F.7.4 Calls-Relation

- **CLEAR-CALLS-RELATION-MANAGER**
DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".
- **CALLS-RELATION-TUPLE-EXISTS**
DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if a calls tuple exists in the mgr. Second, if the calls tuple exists, it also updates the pointer to the proper address. The local variable "local-pointer" is necessary because the instantiated package Calls-Relation-Manager-Package expects ptrs of type "Iterator-Type" as parameters to its procedures and functions, thus a type conversion using "local-pointer" is required.
- **CREATE-CALLS-RELATION-TUPLE**
DESCRIPTION: This procedure creates a new calls tuple in the manager. If a calls tuple with the same key does not already exist, then it simply creates a new element. "Calls-Relation-Pointer" is used to pass back to the calling procedure the address of the new calls tuple. Note the pointer is mode "in out" because it is read when passed as a parameter in "Calls-Relation-Tuple-Exists".
- **KILL-CALLS-RELATION-TUPLE**
DESCRIPTION: This procedure simply removes a calls tuple in the mgr at the address of the "Calls-Relation-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse.
- **KILL-CALLS-RELATION-TUPLES-WITH-ACTIVITY-NAME**
DESCRIPTION: This procedure removes all calls tuples that have a specific activity name. This procedure does not interact with the "Historical-Activity-Manager" at all.
- **CALLS-RELATION-TUPLE-WITH-ACTIVITY-NAME-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of one or more tuples with a specific activity name. The function stops when the first tuple with the correct name is found.
- **VALUE-OF-CALLS-RELATION-TUPLE**
DESCRIPTION: There are two functions that can retrieve a calls tuple. "Value-Of-Calls-Relation-Tuple-At-Iterator", however, is for use only in conjunction with the other iterator functions and procedures. The purpose of this function is to retrieve the entire tuple at the location provided by the pointer. Note that because the entire tuple is the key, the function is not really necessary since you must supply the tuple to the "Calls-Relation-Tuple-Exists" procedure before this function. Thus, a purist could claim this function to be "window dressing". However, it is always possible one or more non-key attributes may later be added; therefore, this function would be required given the implementation technique chosen by the author.
- **NUMBER-OF-CALLS-RELATION-TUPLES**
DESCRIPTION: This function returns the number tuples in the manager. A 0 is returned if empty. Thus, this function fills in for the "Calls-Relation-Tuple-Manager-Null?" function. One could argue that it should be included because it would be $O(1)$ vs $O(c)$; but note that if the manager is empty, this function is $O(1)$, so you only sacrifice when its not empty.
- **RESET-CALLS-RELATION-TUPLE-ITERATOR**
DESCRIPTION: This procedure sets the global iterator to point to the first item in the manager.

- **VALUE-OF-CALLS-RELATION-TUPLE-AT-ITERATOR**
DESCRIPTION: This procedure retrieves the item in the manager at the position pointed to by the global (to this body) variable "Calls-Relation-Iterator".
- **ADVANCE-ITERATOR-TO-NEXT-CALLS-RELATION-TUPLE**
DESCRIPTION: This procedure advances the global iterator to point to the next item in the manager.
- **CALLS-RELATION-TUPLE-ITERATOR-DONE**
DESCRIPTION: This function returns a boolean value indicating if the global iterator is pointing to a null (done) position.

F.7.5 Consists-Of

- **CLEAR-CONSISTS-OF-RELATION-MANAGER**
DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".
- **CONSISTS-OF-RELATION-TUPLE-EXISTS**
DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if a Consists-Of tuple exists in the mgr. Second, if the Consists-Of tuple exists, it also updates the pointer to the proper address. The local variable "local-pointer" is necessary because the instantiated package Consists-Of-Relation-Manager-Package expects ptrs of type "Iterator-Type" as parameters to its procedures and functions, thus a type conversion using "local-pointer" is required.
- **CONSISTS-OF-RELATION-TUPLE-WITH-PARENT-CHILD-DE-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one tuple that has the same parent and child data element names. The function stops when the first such tuple is found.
- **CREATE-CONSISTS-OF-RELATION-TUPLE**
DESCRIPTION: This procedure creates a new Consists-Of tuple in the manager. If a Consists-Of tuple with the same key does not already exist, then it simply creates a new element. "Consists-Of-Relation-Pointer" is used to pass back to the calling procedure the address of the new Consists-Of tuple. Note the ptr is mode "in out" because it is read when passed as a parameter in "Consists-Of-Relation-Tuple-Exists". This procedure performs one consistency check. If the parent data element is not the same as the child, the procedure reverses the parent and child names in a temporary tuple and makes sure a tuple doesn't exist. Thus, it prevents 'm' from having a child 'a', if 'm' is already a child of 'a', which is not possible.
- **KILL-CONSISTS-OF-RELATION-TUPLE**
DESCRIPTION: This procedure simply removes a Consists-Of tuple in the mgr at the address of the "Consists-Of-Relation-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse.
- **KILL-CONSISTS-OF-RELATION-TUPLES-WITH-CONSISTS-OF-ID**
DESCRIPTION: This procedure removes all Consists-Of tuples that have a specific consists-of-id. Basically, when one decides to delete, for instance, that 'm' consists of x, y, and z then the first step is to get the consists-of-id and the second step is to execute this operation.
- **VALUE-OF-NEXT-CONSISTS-OF-ID**
DESCRIPTION: This function simply returns a natural value indicates the next available id number for a composite data element when it is decomposed into two or more data elements which themselves can be composite data elements. '1' is added to the previous highest id number and the sum is returned.

- **VALUE-OF-CONSISTS-OF-ID**
DESCRIPTION: This function accepts as input a composite data element name (parent) and a list of data elements (children). From this information the function determines the Consists-Of-Id that has been assigned to this decomposition. Please note the following: If the data elements aren't found a zero value is returned!
- **Consists-Of-RELATION-TUPLE-WITH-PARENT-DE-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one tuple with the given parent data element name. The function stops when the first tuple with a matching parent name is found.
- **Consists-Of-RELATION-TUPLE-WITH-CHILD-DE-EXISTS**
DESCRIPTION: This function simply returns a boolean value that indicates the presence or absence of at least one tuple with the given child data element name. The function stops when the first tuple with a matching child name is found.
- **VALUE-OF-CONSISTS-OF-RELATION-TUPLE**
DESCRIPTION: There are two functions that can return a Consists-Of tuple. "Value-Of-Consists-Of-Relation-Tuple-At-Iterator", however, is for use only in conjunction with the other iterator functions and procedures. The purpose of this function is to retrieve the entire tuple at the location provided by the pointer. Note that because the entire tuple is the key, the function is not really necessary since you must supply the tuple to the "Consists-Of-Relation-Tuple-Exists" procedure before this function. Thus, a purist could claim this function to be "window dressing". However, it is always possible one or more non-key attributes may later be added; therefore, this function would be required given the implementation technique chosen by the author.
- **NUMBER-OF-CONSISTS-OF-RELATION-TUPLES**
DESCRIPTION: This function returns the number tuples in the manager. A 0 is returned if empty. Thus, this function fills in for the "Consists-Of-Relation-Tuple-Manager-Null?" function. One could argue that it should be included because it would be $O(1)$ vs $O(s)$; but note that if the manager is empty, this function is $O(1)$, so you only sacrifice when its not empty.
- **RESET-CONSISTS-OF-RELATION-TUPLE-ITERATOR**
DESCRIPTION: This procedure sets the global iterator to point to the first item in the manager.
- **VALUE-OF-CONSISTS-OF-RELATION-TUPLE-AT-ITERATOR**
DESCRIPTION: This procedure retrieves the item in the manager at the position pointed to by the global (to this body) variable "Consists-Of-Relation-Iterator".
- **ADVANCE-ITERATOR-TO-NEXT-CONSISTS-OF-RELATION-TUPLE**
DESCRIPTION: This procedure advances the global iterator to point to the next item in the manager.
- **CONSISTS-OF-RELATION-TUPLE-ITERATOR-DONE**
DESCRIPTION: This function returns a boolean value indicating if the global iterator is pointing to a null (done) position.

F.7.6 Historical-Activity

- **CLEAR-HISTORICAL-ACTIVITY-MANAGER**
DESCRIPTION: This procedure simply clears the manager and sets the iterator to "null".

- **HISTORICAL-ACTIVITY-EXISTS**
DESCRIPTION: This procedure serves 2 functions. First it informs the user via the "Found-Flag" if a historical activity exists in the mgr. Second, if the historical activity exists, it also updates the pointer to the proper address. The local variable "local-pointer" is necessary because the instantiated package Historical-Activity-Manager-Package expects ptrs of type "Iterator-Type" as parameters to its procedures and functions, thus a type conversion using "local-pointer" is required.
- **CREATE-HISTORICAL-ACTIVITY**
DESCRIPTION: This procedure creates a new historical activity in the manager. If a historical activity with the same key does not already exist, then it simply creates a new element and assigns the key. "Historical-Activity-Pointer" is used to pass back to the calling procedure the address of the new historical activity.
- **KILL-HISTORICAL-ACTIVITY**
DESCRIPTION: This procedure simply removes a historical activity in the mgr at the address of the "Historical-Activity-Pointer". The key feature of this procedure is that the element pointer is returned with a null value which prevents potential user abuse.
- **VALUE-OF-HISTORICAL-ACTIVITY**
DESCRIPTION: There are two functions that can retrieve a historical activity tuple. "Value-Of-Historical-Activity-At-Iterator" however is for use only in conjunction with the other iterator functions and procedures. The purpose of this function is to retrieve the entire tuple at the location provided by the pointer. Note that because the entire tuple is the key, function is not really necessary since you must supply the project name and number to the "Historical-Activity-Exists" procedure before this function. Thus, a purist could claim this function to be "window dressing". However, it is always possible one or more non-key attributes may later be added; therefore, this function would be required given the implementation technique chosen by the author.
- **NUMBER-OF-HISTORICAL-ACTIVITIES**
DESCRIPTION: This function returns the number of historical activities in the manager. A 0 is returned if empty. Thus, this function fills in for the "Historical-Activity-Manager-Null?" function. One could argue that it should be included because it would be $O(1)$ vs $O(h)$; but note that if the manager is empty, this function is $O(1)$, so you only sacrifice when its not empty.
- **RESET-HISTORICAL-ACTIVITY-ITERATOR**
DESCRIPTION: This procedure sets the global iterator to point to the first item in the manager.
- **VALUE-OF-HISTORICAL-ACTIVITY-AT-ITERATOR**
DESCRIPTION: This procedure retrieves the item in the manager at the position pointed to by the global (to this body) variable "Historical-Activity-Iterator".
- **ADVANCE-ITERATOR-TO-NEXT-HISTORICAL-ACTIVITY**
DESCRIPTION: This procedure advances the global item iterator to point to the next item in the manager.
- **HISTORICAL-ACTIVITY-ITERATOR-DONE**
DESCRIPTION: This function returns a boolean value indicating if the global iterator is pointing to a null (done) position.

Bibliography

1. Armstrong, J.R. *Chip-Level Modeling with VHDL*. Englewood Cliffs NJ: Prentice Hall, 1989.
2. Austin, Kenneth A. *Structured Analysis Tool Interface to the Strategic Defense Initiative Architecture Dataflow Modeling Technique*. MS thesis, AFIT/GCS/ENG/88D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A126281).
3. Austin, Kenneth A. and others. "An Entity Relationship Modeling Approach to IDEF₀ Syntax," *Proceedings of IEEE 1990 National Aerospace and Electronics Conference NAECON 1990*, 2:641-645 (May 1990).
4. Bailor, Maj Paul D. "Verification and Validation of Real-Time Software." Class hand-out for CSCE 693, Principles of Embedded Software, Winter Quarter 1991.
5. Balzer, R. "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, SE-11(11):1257-1268 (November 1985).
6. Balzer, R. and others. "Software Technology in the 1990s: A New Paradigm." *IEEE Computer*, 16(11):39-45 (November 1983).
7. Barton, David L. "Behavioral Descriptions in VHDL," *VLSI Systems Design*, pages 28-33 (1988).
8. Barton, David L. "A First Course in VHDL," *Design Automation Guide*, pages 40-47 (1988).
9. Blankenship, Donald D. *Generalized Method for Transforming Informal Analysis Methods to a Refine Formal Specification*. MS thesis, AFIT/GCS/ENG/91D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
10. Boehm, Barry W. "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, 1(1):75-88 (January 1984).
11. Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.
12. Brooks, Frederick P., Jr. "No Silver Bullet," *IEEE Computer*, 20(4):10-19 (April 1987).
13. Connor, Michael F. "SADT - Structured Analysis and Design Technique Introduction." In *1980 International Engineering Management Conference Record*, pages 138-143, New York: IEEE Press, 1980.
14. Davis, Alan M. *Software Requirements : Analysis and Specification*. Englewood Cliffs NJ: Prentice Hall, 1990.
15. Davis, J. S. "Identification of Errors in Software Requirements Through the Use of Automated Requirements Tools," *Information and Software Technology*, 31(9):472-476 (November 1989).

16. Douglass, Randall L. *Formalization and Validation of an SADT Specification Through Executable Simulation Using the Refine Specification Environment*. MS thesis, AFIT/GCS/ENG/91D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
17. *Fourth International Workshop on Software Specification and Design*, Washington, DC: Computer Society Press of the IEEE. April 1987.
18. Freeman, P. "Requirements Analysis and Specification: The First Step." In *Proceedings of the International Computer Technology Conference*, pages 290-296, San Francisco: ASME, August 1980.
19. Godwin, Anthony N. and others. "An Assessment of the IDEF Notations as Descriptive Tools," *Information Systems*, 14(1):13-28 (1989).
20. Harel, David and others. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, 16(4):403-414 (April 1990).
21. Hartrum, Thomas C. "IDEF₀ Requirements Analysis." Class handout describing the use of IDEF₀ for software requirements analysis, October 1989.
22. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards (Draft 4 Edition)*. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, January 1989.
23. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures in Pascal: Third Edition*. New York: Computer Science Press, Inc., 1990.
24. Hurley, Richard B. *Decision Tables In Software Engineering*. New York, NY: Van Nostrand Reinhold Company, Inc., 1983.
25. IEEE Press, New York. *IEEE Standard VHDL Language Reference Manual - IEEE Std 1076-1987*, 1988.
26. Kellner, Mark J. "Position Paper: Software Process Modeling at SEI." In *Proceedings of the Conference on Software Maintenance*, page 78, New York: IEEE, October 1988.
27. Kitchen, Terry L. *An Object-Oriented Design and Implementation for the IDEF₀ Essential Data Model with an Ada Based Expert System*. MS thesis, AFIT/GCS/ENG/90D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230814).
28. Kramer, Jeff and Keng Ng. "Animation of Requirements Specifications," *Software Practice and Experience*, 18(8):749-774 (August 1988).
29. Langloss, Randel K. *Graph-Based Visualization of Formal Specification and Domain Specific Languages*. MS thesis, AFIT/GCS/ENG/91D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
30. Levi, Shem-Tov and Ashok K. Agrawala. *Real-Time System Design*. New York: McGraw-Hill Book Company, 1990.

31. Lipsett, Roger and others. *VHDL: Hardware Description and Design*. Boston: Kluwer Academic Publishers, 1989.
32. Marca, David A. and Clement L. McGowan. *SADT Structured Analysis and Design Technique*. New York: McGraw-Hill Book Company, 1988.
33. Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH 45433. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF₀)*, June 1981. Contract F33615-78-C-5158 with SofTech, Inc.
34. Metzner, John R. and Bruce H. Barnes. *Decision Table Languages and Systems*. New York, New York: Academic Press, 1977.
35. Miller, Richard L. *Specification and Equivalence Verification of Sequential Circuits via VHDL*. MS thesis, AFIT/GE/ENG/90D-41, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1990 (AD-A230554).
36. Pollack, Solomon L. *Decision Tables: Theory and Practice*. New York: John Wiley and Sons, Inc, 1971.
37. Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Book Company, 1987.
38. Reasoning Systems Inc. *Refine User's Guide*, 1985. Version 3.0 Beta, revised 1989.
39. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, 1:16-34 (January 1977).
40. Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Computer*, 18(4):25-35 (April 1985).
41. Ross, Douglas T. and K. Schoman. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, 1:6-15 (January 1977).
42. Schach, Stephen R. *Software Engineering*. Homewood, IL: Richard D. Irwin Inc, and Asken Associates Inc., 1990.
43. Shyong, Min-fuh. *An Ada Based Expert System for the Ada Version of SAtool II*. MS thesis, AFIT/GCS/ENG/91J-XX, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1991 (AD-A238887).
44. Smith, Nealon F. *SAtool II: An IDEF₀ Syntax Data Manipulator and Graphics Editor*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215289).
45. Smith, Sharon L. and Susan L. Gerhart. "STATEMATE and Cruise Control: A Case Study." In *Proceedings of the Twelfth Annual International Computer Software and Applications Conference (COMPSAC 88)*, pages 49-56, New York: IEEE, October 1988.
46. Sommerville, Ian. *Software Engineering*. Workingham, England: Addison-Wesley, 1989.

47. Teichroew, Daniel and Earnest A. Hershey III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, 3(1):41-48 (January 1977).
48. Tevis, Jay-Evan J. *An Ada-based Framework for an IDEF₀ CASE Tool Using the X Window System*. MS thesis, AFIT/GCS/ENG/90D-15, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A231239).
49. Topper, Andrew. "Automating Software Development," *IEEE Spectrum*, 28(11):56-62 (November 1991).
50. Webster, Dallas E. "Mapping the Design Information Representation Terrain," *IEEE Computer*, 21(12):8-23 (December 1988).
51. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs NJ: Prentice-Hall, 1989.
52. Zave, Pamela. "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, SE-8(3):250-269 (May 1982).
53. Zave, Pamela and William Schell. "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transactions on Software Engineering*, SE-12(2):312-326 (February 1986).
54. ZYCAD Corporation. *ZYCAD System VHDL Reference Manual*, 1990. REV 2.0a.
55. ZYCAD Corporation. *ZYCAD System VHDL User's Manual*, 1990. REV 2.0a.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204 Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1991	3. REPORT TYPE AND DATES COVERED Master's Thesis		
4. TITLE AND SUBTITLE FORMALIZATION AND VALIDATION OF AN SADT SPECIFICATION THROUGH EXECUTABLE SIMULATION IN VHDL		5. FUNDING NUMBERS		
6. AUTHOR(S) Daniel L. Eickmeier, Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/91D-6		
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING MONITORING AGENCY REPORT NUMBER		
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) Formalizing an informal requirements specification, such as SADT, and executing the formal specification in a simulation environment, such as VHDL, provides a requirements analyst a means to validate the behavior of a specification early in the development life cycle. This research effort investigated and demonstrated the feasibility and benefit of transforming an SADT specification of a system into an equivalent VHDL executable simulation. Both non-time related behavior and concurrent, real-time related behavior is addressed. First, a decision table extension to SADT is created so that detailed, executable behavior can be specified. Next a mapping from SADT to VHDL is defined. Last, this mapping was applied to two example problems: the Heating System and the Lift (Elevator) Control System. An SADT specification was generated for each of these problems, and the resulting specification was transformed into an equivalent VHDL specification using the mapping technique defined by this research. The VHDL simulation environment was used to execute the specification, determine its behavior, make necessary changes, and re-execute the specification until the proper system behavior was specified. The result was an unambiguous specification which can serve as a formal basis for subsequent design and implementation phases, and ultimately, a product which better satisfies the users needs.				
14. SUBJECT TERMS Computer Programs, Computerized Simulation, Prototypes, Requirements, Software Engineering, Specifications, Systems Engineering, VHDL			15. NUMBER OF PAGES 314	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	