1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

Si il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui forment déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

**Adapting an Iconic Software Design Environment
for Protocol Specification with Estelle**

by

Trevor W. Pearce

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Engineering

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University

August, 1986

© copyright

1986, Trevor W. Pearce

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-33417-7

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis,

"Adapting an Iconic Software Design Environment for
Protocol Specification with Estelle"

submitted by Trevor W. Pearce, B. Eng., in partial fulfilment of
the requirements for the degree of Master of Engineering.

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

September, 1986

ABSTRACT

The creation of initial software designs directly from formal protocol specifications saves development time and reduces the potential for errors. This thesis builds on the Caede environment to investigate the translation of protocols specified in the Estelle formal description language into Ada software designs. A unique graphical notation for Estelle is defined and an Ada execution model is shown to be directly generated from the specification diagrams. Application of the notation to the RMS case study shows that the alignment of the protocol specifications and software designs is at least conceptually possible within the framework of Caede. The work undertaken in the thesis has provided insight into issues and directions to be addressed in future work.

ACKNOWLEDGEMENTS

I would like to thank my supervisor and mentor, Ray Buhr for his generous supervision and positive criticism during the entire project. His patience and effort were especially appreciated during the writing phase. I hope to have the opportunity to continue working with him in the future.

I must also thank Murray Woodside for valuable ideas and feedback. The opportunity to co-author a paper would not have materialized without his enthusiasm and direction.

I am in debt to Carol Hayes and Ron Casselman for their hard work and expertise in developing Caede. This thesis would not have been completed without their support. Thanks.

A patient ear and constructive feedback from Gerald Karam have been invaluable. I would like to wish him every success with the future and thank him for support on this project.

Thanks are also due to John Chinneck for philosophical wisdom and ornithological expertise.

I am particularly thankful to Bernie Pagurek for taking a chance and making this thesis and a new future possible.

Family support has been exceptional during the summer months. I hope I will never put my wife and children through this again. I would like to dedicate this thesis to my parents, thanks for all the help.

TABLE OF CONTENTS

1) Introduction	1-1
1.1 Introduction	1-1
1.2 Background	1-3
1.3 Statement of Thesis	1-6
1.4 Project History and Author's Work	1-7
1.5 Layout of Thesis	1-10
2) Caede/Estelle	2-1
2.1 Graphical Representation	
of Estelle in Caede	2-1
2.1.1 Icons for Estelle	2-2
2.1.2 Channel Template Diagrams	2-3
2.1.3 Structure Diagrams	2-4
2.1.4 State Machine Diagrams	2-8
2.1.5 Mapping the Notation to Estelle	2-12
2.2 Caede Methodology for Estelle Specifications	2-17
2.2.1 Top Down	2-18
2.2.2 Bottom Up	2-19
2.2.3 Automated Support	2-20
3) Case Study: RMS Example	3-1
3.1 Caede Diagrams	3-3
3.2 Estelle Specifications	3-5
4) Execution of Estelle Specifications	4-1
4.1 Execution Issues	4-1
4.1.1 Hierarchy Execution Rule	4-2

4.1.2 Non-Determinant Choices	4-3
4.1.3 Exported Variables	4-4
4.2 Dispatcher/Worker Model	4-4
4.2.1 Dispatcher Task	4-6
4.2.2 Worker Tasks	4-9
4.2.3 Channels	4-9
4.2.4 Modules	4-10
4.3 Automatic Code Generation for Caede	4-10
4.3.1 Dispatcher Task	4-11
4.3.2 Worker Tasks	4-12
4.3.3 Channels	4-13
4.3.4 Modules	4-15
4.3.5 State Machines	4-17
5) Discussion	5-1
5.1 Enhancements to the Graphical Notation for Estelle	5-1
5.1.1 Representation of Module Sub-structure	5-2
5.1.2 Dynamic Operations on Objects	5-4
5.1.3 Process and Activity Modules	5-6
5.1.4 Textual Information	5-7
5.2 Broader Graphical Issues	5-8
5.2.1 Hierarchical Dependency	5-8
5.2.2 Direct Coupling and Abstraction	5-10
5.2.3 Switch Modules and Abstraction	5-12
5.3 Execution Models for Estelle	5-13
5.3.1 Task-Per-Process: Version 1	5-14
5.3.2 Task-Per-Process: Version 2	5-16

5.3.3 Task-Per-Process: Version 3	5-18
5.3.4 Justification of Tasks	5-19
5.4 Mapping of Estelle Specifications into	
Software Designs	5-20
5.2 Enhancements to Caede's Automated Support	5-30
5.2.1 Side Effects of Diagram Modifications	5-31
5.2.2 Views	5-32
6) Conclusions	6-1

References

LIST OF FIGURES

Figure	Title	Page
2-1	Icons for Estelle	2-23
2-2	Channel Template Structure	2-24
2-3	U_ACCESS_POINT Channel Template	2-25
2-4	Connected Alternating Bit Protocol Modules	2-26
2-5	Internal Structure of ALT_BIT_2 Module	2-27
2-6	Module Structure with State Machine and Nested Modules	2-28
2-7	View of States and Transitions for ALT_BIT_1 State Machine /	2-29
2-8	Coupled ALT_BIT_1 State Machine	2-30
2-9	Text Specification Format of Estelle Module	2-31
2-10	Mapping of Caede Diagram Contents to Estelle Text	2-32
2-11	Alternating Bit Protocol Modules	2-33
2-12	Channel Templates for Alternating Bit Protocol	2-34
2-13	RX Module State Machine	2-35
2-14	Unconnected ALT_BIT_2 Modules	2-36
2-15	RX_U_CHNL Channel Template	2-37
2-16	View of Transitions Out of ESTAB State	2-38
3-1	RMS for Connected Nodes	3-10
3-2	Architecture for a Single Node	3-11
3-3	RMS Architecture for a Node	3-12
3-4	Channels for SERVICE Interaction Points	3-13
3-5	SERVICE Module Internal Structure	3-14

3-6	SOURCE Module Internal Structure . . .	3-15
3-7	Module State Machine . . .	3-16
4-1	Example of Process Hierarchy in a Specification . . .	4-24
4-2	Architecture of the Dispatcher/Worker Model	4-25
4-3	Internal Structure of a Channel Package .	4-26
4-4	Internal Structure of a Module Package .	4-27
5-1	Internal Structure of LINK Module . . .	5-34
5-2	Direct Coupling of SEND and S_QUEUE Modules	5-35
5-3	Example of the Task-Per-Process Execution Model . . .	5-36
5-4	Active Channels for the Task-Per-Process Model . . .	5-37
5-5	Modified SEND State Machine . . .	5-38

LIST OF APPENDICIES

Appendix A	Estelle Specification of the Alternating Bit Protocol
Appendix B	Estelle Background Relevant to Caede/Estelle Diagrams

1. Introduction

1.1) Introduction

The complexity of communication systems increases almost daily as the technology to communicate farther and faster is explored. The advantages of new techniques and technologies can only be realized through cost effective and reliable implementations. Formal description techniques and models have been developed to allow new communication protocols to be unambiguously introduced and integrated into communication systems. Simultaneously, the development of software engineering techniques have enabled increasingly more complex systems to be implemented reliably, using progressively more abstract design tools.

The motivation for this thesis is the alignment of the protocol specification process and the software engineering process. An ideal alignment would produce a software design using the protocol specification as a framework. This approach takes advantage of testing and verification work already done for the protocol to reduce overall time, effort and opportunity for error in developing the software design. In contrast, a design constructed without a framework provided by the specification must re-iterate through the verification of the structure and behaviour of the protocol portions before progressing to the implementation dependent aspects.

The starting point for this work was the aim to use the Caede [1,2] environment to align protocols specified in the Estelle [3] formal description language with software designs in the Ada [4] programming language. The integration of the protocol specification process into Caede would utilize a graphical interface to create an initial Estelle specification. The resulting diagrams would then provide the framework from which the Ada software design evolved. Ideally, executable constructs would be automatically generated from the graphics at both the protocol testing/verification and target implementation stages.

This thesis focuses on a graphical notation for Estelle and the execution of Estelle specifications. The intended result of this thesis is to provide a "proof-of-concept" for representing Estelle graphically and the eventual alignment of protocol specification and software design using Caede. Issues are identified and directions are suggested for future work towards the complete alignment.

For the purposes of this document, the terms specification and design are used consistently to mean the description of protocols and software respectively. A protocol specifier creates a protocol specification which is implementation independent. A software designer produces software designs which are implementation dependent. From a life cycle perspective, protocol specifications are normally created prior to preparing software designs.

1.2) Background

The use of graphics to represent protocols and software is fundamental to this thesis. A short overview of Caede's current capabilities provides a baseline for comparison to similar projects. A detailed introduction to the project and the author's work is given later in section 1.4.

Research activities related to this thesis involve formal description techniques, graphical notations, specification tools and the alignment of the specification process with software engineering. The formal description techniques for protocol systems include state machines [5], Petri nets [6], and formal algebra [7]. All of these methods have been implemented using graphical notations [5,8,9] and have been integrated into specification tools [10,8,9]. Some success has been achieved in aligning graphically represented specifications with software designs. To the author's knowledge, the approach taken in this thesis of graphical representation of Estelle in Caede and alignment with Ada software designs is unique.

Engineers like to use diagrams as a medium for recording and communicating information. Fairley [11] surveys some graphical notations for software engineering which are applicable in a wide range of applications. The Structured Analysis and Design Technique (SADT) may be used to depict complex requirements concisely. The Structured System Analysis (SSA) notation is suited to data processing applications. The Requirements

Statement Language (RSL) provides a data flow oriented graphical representation for real-time process control systems. Other commonly used notations described by Fairley include data flow "bubble charts", HIPO (Hierarchy Input Process Output) diagrams and flowcharts.

Caede's interactive graphical interface allows sets of icons to be combined into diagrams which represent software structure and inter-object interactions. Caede's current software model is Ada oriented and the supporting notation includes icons for tasks, packages, rendezvous, procedures and data. Details of the use of Caede's notation for software engineering and the associated human interface may be found in [12,2].

Caede is more than an intelligent graphical editor. Support tools [13] allow the structure diagrams to be analyzed for deadlock and performance characteristics. Ada code may also be generated automatically for the structures and interactions identified in the diagrams.

Although early work towards Estelle [14,15] frequently used state machine diagrams to express protocols, the current ISO standard for Estelle [3] does not include a graphical notation. The extension of Caede's notation described in this thesis was originated by the Caede project team to allow graphical representation of Estelle. New icons and diagrams have been introduced to represent state machines in the framework of Estelle.

modules.

Other work on Estelle oriented design environments includes the multi-national European SEDOS Project [16]. Estelle tools being developed for SEDOS include a syntax driven editor, a compiler and a debugger. One of the objectives of the project is to gain experience to be applied towards future research into automated preparation of protocol implementations.

SDL [5] is another state machine based formal description technique. The language was proposed by the CCITT and includes a graphical notation which represents a state machine as a collection of drawings. Each drawing shows all of the transitions which originate from a particular state. A single diagram showing all possible transitions between states is never constructed.

SDL has been successfully incorporated into the Melba project [10]. Melba supports interactive graphical specification of protocols using the SDL notation. The specification is then translated directly to the CHILL language.

Numerical Petri Nets (NPN) [8] are an extension to Petri nets [6]. Although NPN's do not contain a formal graphical notation, Petri nets are fundamentally graphic in nature. PROTEAN [8] is a protocol specification tool which accepts textual definitions of NPN's and then graphically represents the nets and various analysis results.

PROT nets [17] (another extension to Petri nets) are currently being researched in Italy with the goal of rapid prototyping of process control systems. Existing PROT net development tools include an interactive graphical editor and a translator which creates Ada skeleton code directly from the net diagrams.

Formal algebra is also been used to specify protocols. Lotos [18] uses a language based on the Calculus for Communicating Systems (CCS) [19]. Work is currently being undertaken [20,21] to develop Lotos execution and testing tools. The author is unaware of any graphical notations which directly support Lotos specifications, however IDCCS [9] supports the graphical representation of CCS.

1.3) Statement of Thesis

It is possible to represent the framework of Estelle specifications graphically with Caede. Not all aspects of Estelle are suited to graphical representation but sufficient aspects may be modeled graphically to generate the framework of an Estelle specification automatically. Information not suited to graphics must be input as text. It is also possible to generate an Ada software design automatically from the Caede graphics and textual input to execute an Estelle specification. Therefore, Caede provides a framework for the alignment of protocol specifications and software designs.

1.4) Project History and Author's Work

Caede has been developed by a design team. Much of the framework for Caede was in place before the author joined the team. This section presents the project background related to the author's work. The development of the project is presented chronologically in the same sequence as the author's contributions. The contributions of this thesis are identified where appropriate.

One of the original objectives for the Caede project was the integration of a graphical notation for the representation of protocols into an interactive design environment. Early versions of Caede (developed prior to the author's involvement) addressed the mechanics of an interactive graphical interface, the representation of the structure of software designs and the integration of support tools. The author has provided contributions in the more recent developments of the notation for representing temporal information as state machines and the specification of Estelle graphically. For future reference, the work involving the author will be referred to as Caede/Estelle to differentiate it from other developments and activities in Caede.

The development of Caede has been carried out by a design team headed by Dr. R.J.A. Buhr. Caede/Estelle includes contributions by Dr. Buhr, Dr. C.M. Woodside, Gerald Karam and the author. Implementation of the notation in Caede has been carried

out by Carol Hayes and Ron Casselman. The remainder of this section describes work to which the author made major contributions.

As previously discussed, the original motivation for Caede/Estelle was the alignment of protocol specifications with software designs in the framework of Caede. The selection of Estelle as the protocol specification language was based on the significance of Estelle as a proposed ISO standard. Furthermore, early drafts of the Estelle language [22] appeared to have convenient mappings to the existing Caede icons. Ada was selected as the execution language due to the existing Ada orientation of the Caede software design icons.

Early work [23] developed an initial state machine notation modeled after the early version of Estelle. The notation was applied to the RMS protocol [24] and successfully represented both the protocol specification and implementation dependent details. The conclusions of the work indicated that using the notation for the alignment of protocol specification and software engineering appeared consistent with Estelle.

The next release of the Estelle language [3] contained extensions to the initial structural concepts including the introduction of module types (process and activity) and dynamic operations on objects. The mapping of the state machine notation to the new version of Estelle was consistent with previous work.

Some of the new extensions to Estelle were not described by the notation but the notation was still considered complete enough to prove the concept of representing Estelle graphically.

The integration of the notation into Caede required the development of methodologies for the specification of protocols using the notation. Both top-down and bottom-up approaches were successfully applied manually to the alternating bit protocol and the top-down methodology was then implemented in Caede.

This thesis provides a detailed description of the current Caede/Estelle notation and the methodologies for using the notation in Caede. Caede/Estelle was first documented in [25]. A refined version of the report [26] was presented at the Sixth International Workshop on Protocol Specification, Testing and Verification in June, 1986.

The implementation of the notation into Caede was tested by specifying the RMS protocol graphically [27].

This thesis contains a description of the specification of the RMS Service layer using the notation and then illustrates a skeleton of the Estelle specification which could be generated from the diagrams.

The execution of Estelle using an Ada software design was attempted next. The resolution of issues in the execution behaviour of Estelle resulted in the Dispatcher/Worker execution

model. The automatic generation of code to construct the components in the model was also investigated.

This thesis contains a description of the Dispatcher/Worker execution model and a discussion of the issues associated with the direct execution of Estelle specifications. Guidelines for automatic code generation for the model are also presented.

This thesis identifies and discusses some of the issues which were uncovered while attempting to align the protocol specification and software engineering processes in Caede. The presentation of the issues is intended to identify directions for future work.

1.5) Layout of Thesis

The first chapter has introduced and stated the thesis. Chapter 2 defines the graphical notation for Estelle and illustrates two methodologies for using the notation in Caede. Chapter 3 applies the notation to the RMS example. Chapter 4 deals with the execution of Estelle specifications and presents guidelines for automatic code generation. Chapter 5 is a general discussion section. Issues raised in earlier chapters are discussed and suggestions are made for future work. The chapter includes the identification of issues in the alignment of specifications with implementation designs. Chapter 6 provides conclusions.

Appendix A contains the text for the Estelle specification of the alternating bit protocol. Diagrams representing the specification are used to illustrate the graphical notation in Chapter 2. Appendix B presents a brief summary of some of the concepts underlying Estelle. The sub-sections in the appendix parallel the discussion of constructing Caede diagrams in section 2.1.

Where applicable, diagrams are included at the end of each chapter.

2. Caede/Estelle

The description of Caede/Estelle is broken into two parts. Section 2.1 describes the graphical notation for Estelle. Section 2.2 illustrates how Caede might be used in both top-down and bottom-up methodologies. Throughout both sections, an already existing Estelle text specification of the alternating bit protocol [22, included for convenience in Appendix A] is used as the springboard for explanations.

2.1) Graphical Representation of Estelle in Caede

Caede represents Estelle information by a set of icons which may be assembled in three basic diagram types. Channel template diagrams describe Estelle channel types. Structure diagrams show module hierarchy and interconnection. State machine diagrams depict temporal information.

This section introduces the icon set and the combination of icons into diagrams. Channel templates are discussed first followed by structure diagrams and state machine diagrams. The description of the diagrams is presented in terms of diagram contents and rules for combining icons. The details of the Caede human interface commands to select and manipulate icons have not been included. The section concludes with a summary of the mapping of the notation to the Estelle language.

A working knowledge of Estelle is assumed in the discussion of the representation of Estelle graphically. For the convenience

of readers unfamiliar with Estelle, Appendix B includes brief introductions to the Estelle concepts being modeled by the diagram types. The sections of Appendix B parallel the three diagram types.

2.1.1) Icons for Estelle

The icons developed for the Estelle notation are shown in Figure 2-1. This description provides identification and a brief statement of the intended use for each icon.

The fundamental structural element in Estelle is a module which is represented by the "black box" in Figure 2-1a). There is currently no graphical distinction between process and activity modules. The need for graphical differentiation of the module types is discussed later in chapter 5.

Modules communicate through channels. Channel types are specified in channel template diagrams which are discussed later. Communication between two specific modules is done with an instance of a channel which is drawn as the double headed arrow shown in Figure 2-1b).

A module interaction point exists when a channel is connected to a module. An interaction point is graphically represented by the interface port shown in Figure 2-1c).

Communication primitives determine the communication supported by a channel and are represented by the port plugs shown

in Figure 2-1d) and e). The interpretation of the solid and hollow plugs is left for the discussion of constructing the various diagram types.

The dotted arrow shown in Figure 2-1f) is used on structure diagrams to represent the "attach" operation.

State machines consist of states and transitions. States are represented by hollow circles as shown in Figure 2-1g) while the special double circle of Figure 2-1h) represents an initial state for the state machine. Transitions between states are represented by the double arrow shown in Figure 2-1i).

The construction of state machine diagrams requires the existence of the "transition connection points" and event coupling icons shown in Figures 2-1j) and k) respectively. Details of use of these icons will be dealt with in section 2.1.4.

2.1.2) Channel Template Diagrams

Channel template diagrams specify channel types. All channel template diagrams are based on the initial structure shown in Figure 2-2. The double headed arrow is used to signify bidirectional communication between the two modules which may be connected with the supporting roles. The roles are represented by the two labeled interface ports and Caede assigns the default labels of "user" (u) and "provider" (p).

The construction of a unique channel template requires the specifier to assign a label which identifies the channel type being depicted. The primitives may then be added to the appropriate roles. A primitive is added as a solid port plug (Figure 2-1d) and is placed on the interface port corresponding to the role which allows initiation of the primitive. The specifier must enter a label to identify the primitive when the port plug is added.

Figure 2-3 shows an example of a channel template. The diagram specifies the U_ACCESS_POINT channel type (see the U_ACCESS_POINT channel type in Appendix A). A module which is assigned the provider role may initiate the S_response primitive. The module must also be capable of processing the R_request and S_request messages from the provider role queue when the primitives are activated by a module assigned to the user role.

2.1.3) Structure Diagrams

Structure diagrams show the interactions between a parent module and its immediate children. The parent module is drawn as the outer border of the diagram with external interaction points represented as interface ports on the boundary. The port plugs associated with the connected channels are exposed on the interface ports. Child modules (with appropriate external interface ports) are drawn within the parent boundary. Port plugs are not exposed for child module interface ports. Channel instances are drawn between pairs of interface ports.

The construction of a structure diagram begins with an empty parent boundary and its external interface ports. Child modules are added first by selecting a location inside the boundary, scaling the size of the module icon (Figure 2-1a) appropriately and assigning an identifier. Interface ports may then be added to the child by placing the open side of an interface port icon (Figure 2-1c) against the side of the module icon. The interface port must be scaled to a convenient size and then assigned an identifier.

Once a pair of interface ports needed to support a channel have been added then a channel instance (Figure 2-1b) may be drawn to connect two child modules. The process of creating a channel requires the specifier to indicate the port which will support the provider role, the port which will support the user role, assign an identifier and select the channel type to be created. The channel may be drawn with as many bends as necessary to conveniently connect the two ports.

The attach icon (Figure 2-1f) may be added once the appropriate child module interface port has been added. The arrow is drawn from the child interface port into the parent port which is to be bound. As with channel icons, the arrows may be bent as required.

The structure of a system using the alternating bit protocol is shown in Figure 2-4. The parent module is labeled "SYSTEM" and

would represent a "specification" module. The USER, TIMER, ALT_BIT_2 and NETWORK modules are nested as direct child modules of system. The "U" channel is an instance of the U_ACCESS_POINT type shown previously in Figure 2-3 and has been connected between the U_user interface port on the USER module and the U_provider interface port on the ALT_BIT_2 module. The creation of the U channel required identifying the U_provider interface port to support the provider role, the U_user port to support the user role, assigned the label "U" and selecting the U_ACCESS_POINT channel type. The "S" channel shows a channel icon which has been drawn with a bend to conveniently connect the ports.

The existence of a channel instance automatically determines the plugs which are present on interface ports. The channel template role information is used to determine whether primitives should be represented as solid or hollow plugs (Figure 2-1d and e). The interpretation of solid/hollow plug icon is based on how the primitives are viewed by a connected module. Solid plugs represent primitives which may be initiated by the module connected at a specific port. Hollow plugs indicate primitives which are initiated by the module at the other end of the channel. The plugs are exposed when the port is displayed as an external interface port on the boundary of structure (and state machine) diagrams.

The internal structure of the ALT_BIT_2 module is shown in Figure 2-5. (Note that this diagram does NOT reflect the structure of the alternating bit module presented in Appendix A.

The diagram is used here only to illustrate ports and plugs while the reason for the difference in structure is discussed in Section 2.2.) The external interface ports of ALT_BIT_2 have been drawn with the appropriate plugs exposed. The U_provider port is connected (on Figure 2-4) with an instance of the U_ACCESS_POINT channel type defined in Figure 2-3. The port has been connected with the provider role and consequently the solid plug indicates ALT_BIT_2 may initiate the R_response primitive. The hollow plugs indicate ALT_BIT_2 must service the S_request and R_request primitives when the primitives are initiated by the module at the other end of the channel. The attach icon is used to indicate that the U_SWITCH.U_provider port is attached to the ALT_BIT_2.U_provider port.

Special consideration must be given to the case where the parent module contains a state machine as well as nested sub-modules. The state machine is placed in a black box (identical to the nested sub-modules) and must be given the same identifier as the parent module with the "_FSM" suffix appended. Representing the state machine as a module allows internal interaction points to be added to the state machine as interface ports. Channel connections between the parent and child modules can then be represented consistently. The use of the naming convention to distinguish a state machine from nested modules is discussed further in Chapter 5.

Figure 2-6 shows a module structure which contains both a

state machine and nested modules. The diagram has been created for illustrative purposes only and is not taken from case study work. The parent module (MOD_A) contains two nested modules (MOD_B and MOD_C). The state machine for MOD_A is represented by the MOD_A_FSM box. For MOD_A, the EXT_1_P and EXT_2_U interface ports represent external interaction points while the CH_1_P interface port represents an internal interaction point. The CH_1 channel communicates between the parent module (MOD_A) and the child module (MOD_B). The CH_2 channel communicates between the two child modules.

2.1.4) State Machine Diagrams

Estelle represents temporal behaviour through coupled state machines. State machine diagrams show the interconnection of states by transitions and the coupling of transitions to channel primitives. The states in the machine are represented by state icons (Figure 2-1g and h) and are connected by transition icons (Figure 2-1i). The origin of a transition on the source state is identified by a transition connection point (tcp, Figure 2-1j). The state machine is enclosed in the identical module boundary used for parent module structure diagrams. The boundary identifies the module containing the state machine and provides the interface ports and port plugs which must be coupled to transitions.

As with structure diagrams, the construction of a state machine diagram begins with the boundary of the enclosing module. At this point three options are possible. If a state is added

then the diagram becomes a state machine diagram and no sub-modules may be added. If a module is added first then the diagram becomes a structure diagram and the parent state machine may only be represented by a nested "FSM" module as previously described. If neither a state machine or structure diagram is constructed for a module then the module body is assumed to be defined external to the specification. The remainder of this discussion assumes a state has been added first to create a state machine diagram.

A state is added by selecting the location of the state and assigning an appropriate identifier. A tcp must be placed on the boundary of a state before an associated transition may be added. The transition may not be added until the source and destination states are in place on the diagram. A transition is added by specifying the origin tcp, the destination state and an identifier. As with channel icons, transitions may be drawn containing as many bends as needed to conveniently connect the source and destination state. Transitions may be drawn originating and terminating on the same state.

Figure 2-7 shows the state machine for the ALT_BIT_1 module. This version of the alternating bit module corresponds to the specification given in Appendix A. The module boundary includes the U_provider, T_user and N_user external interface ports. The ESTAB and ACK_WAIT states are present with ESTAB indicated as the initial state. All but the trans_1 and trans_5 transitions originate and terminate on the same states.

The coupling of transitions to channel primitive plugs at the interface ports is represented by the event coupling icon (Figure 2-1k). Couplings may be specified between tcp's and either hollow or solid port plugs. Hollow and solid plugs may be connected to multiple transitions.

Event coupling to a hollow plug may exist only between the plug and the tcp located at the origin of the transition. The coupling indicates that the arrival of the message associated with the primitive is part of the enabling condition for the transition. The coupling is accomplished by connecting the hollow plug to the transition tcp by a solid line. The event coupling icon is then placed at some point parallel to the line and oriented such that it is pointing from the plug towards the tcp. The icon orientation indicates that information flow is from the interface port to the state machine.

A hollow plug may be coupled to multiple transitions. A coupling to a hollow plug is needed for each transition which may be enabled by the reception of the message associated with the primitive.

A transition enabling tcp (located at the origin of the transition) may only be connected to one hollow plug. If a transition is dependent on the arrival of more than one message then additional states are needed to record the arrival of the

messages.

Event coupling to a solid plug indicates that the action block associated with the transition includes the activation of the coupled primitive. The coupling is represented by a solid line connecting a tcp on the transition arrow to the plug. The event coupling icon is placed parallel to the line and oriented to point from the tcp to the plug. The orientation indicates the flow of information is from the state machine to the interface port.

A coupling to a solid plug is needed for each transition which activates the primitive. A transition may be coupled to more than one solid plug if more than one primitive is activated during the action block. The order of tcp's along a transition indicates the order in which the coupled primitives are activated.

Figure 2-8 shows the completely coupled ALT_BIT_1 state machine of Figure 2-7. The arrival of an S_request message (hollow plug) at the U_provider interface port is shown coupled to the trans_1 transition. The coupling indicates that the arrival of an S_request message is part of the enabling condition for the transition. The execution of the trans_1 transition is coupled to the T_user.T_request and a N_user.D_request primitives to propagate information. The order of the tcp's on trans_1 shows the T_request primitive as being initiated before D_request.

Figure 2-8 also shows multiple connections to the `R_request` and `R_response` port plugs. The multiple connections to `R_request` indicate that the `R_request` message is part of the enabling conditions for both the `ESTAB.trans_2` and `ACK_WAIT.trans_2` transitions. Both transitions initiate the `R_response` primitive during transition execution.

2.1.5) Mapping the Notation to Estelle

In this section, a brief description of the textual format for Estelle module specification is followed by a description of how the notation relates to the textual format. The section concludes with a summary of the Estelle text which is currently represented by graphics in Caede. The discussions of whether the current graphical notation is sufficient and future enhancements are postponed until Chapter 5.

The textual contents of an Estelle module specification are represented in the block diagram shown in Figure 2-9. The module header provides interfacing information and defines the visibility of the module at the level it is declared. The module body is hidden from peer modules. The body declaration part defines nested internal sub-modules and channel types as well as constants, types and variables associated with operation. The transition part contains temporal and action information associated with transitions. The initialization and termination parts are procedures which define module creation and termination.

Figure 2-10 shows the relationship between Caede diagram contents and the Estelle textual specification for a module. This diagram forms the basis for the remaining discussion of the mapping. It must be remembered that the current notation deals only with static handling of objects and future dynamic handling will impact the contribution of structure and state machine diagrams to the textual specification. The incorporation of dynamic object handling will influence the representation of text as shown by the dotted lines on Figure 2-10. A more detailed discussion of dynamic handling is given in Chapter 5.

The existence of a module specification is indicated by the presence of a module icon on a structure diagram.

Module Header Information:

The module header information represented on a structure diagram are the identifier and the enumeration of interaction points. The identifier is given by the module icon label and the external interaction points are identified by the interface ports on the module icon boundary. The interface ports drawn on the special "_FSM" module icon represent internal interaction points. Information regarding the roles supported at the interface ports is not represented graphically but is recorded internally in Caede. The assigning of roles to ports is recorded when the channel icon is drawn.

Header information not represented graphically include the module type (process or activity), initialization parameters and exported variable lists.

Module Declaration Part:

The declaration part defines module sub-structuring, types and operational parameters. Structure diagrams contain child module icons to provide identifier and interaction point information for the declaration of nested modules. Channel templates represent channel types. The templates include specification of the type identifier, roles and primitives supported by the roles. Instances of a channel are declared as a variable of the appropriate channel type.

The following information is not represented graphically in the declaration part:

- constants, types, variables, procedures and functions
(same meaning as Pascal)
- parameters associated with channel primitives
- the "use" clause to access exported variables

Module Initialization Part:

The creation of channels and attaching of interface ports must be carried out during the initialization part using the "connect" and "attach" commands. The appropriate interface port and channel icons directly provide the parameters for the

commands. The initialization of the state variable for a state machine is indicated by the initial state icon.

Operational initialization such as variable initialization is not represented graphically. The future representation of dynamic handling of objects graphically will expand the ability to represent "connect", "attach" and other related operations performed in the initialization part.

Module Termination Part:

The termination part is executed before a module is destroyed. The termination part has not been addressed by the notation. The future graphical representation of dynamic handling of objects will impact the representation of the terminate part.

Module Transition Part:

Temporal information is contained entirely in the body transition part and is represented by state machine diagrams. In the certain case described previously, structure diagrams indicate the presence of a transition part for a parent module when an "_FSM" sub-module has been drawn. The details of the state machine are contained in the state machine diagram for the "_FSM" module.

The Estelle syntax corresponding to a graphical transition shown in a state machine diagram is:

```
trans    trans_id
```

from source_state_id
when input_interaction
provided enabling_condition
priority priority_level
transition_block
to destination state_id

Transitions which do not contain a "when" clause are called spontaneous transitions.

The state machine diagrams provide the enumeration of all states and transitions in a state machine. State and transition identifiers are labeled directly on the diagrams. Each graphical transition identifies the "from" and "to" parameters since a transition goes "from" the source state "to" the destination state.

The "when" and "provided" clauses determine the enabling condition for a transition. "When" clauses are represented graphically by the coupling of hollow port plugs to the enabling tcp at the origin of a transition. The parameters of the "when" clause are identified by the interface port and the plug. The existence of a "provided" clause is implied by the enabling tcp. The details of the boolean test associated with the clause are not represented graphically.

The presence of all "out" clauses in transition blocks is represented by couplings between transition tcp's and solid port

plugs.

The following Estelle transition components have not been modeled graphically:

- priority levels
- the "delay" clause has not been considered.

The future, graphical representation of dynamic handling of objects will impact the representation of the transition part since the dynamic manipulation of objects could be carried out during transition block execution.

2.2) Caede Methodology for Estelle Specifications

This section describes general approaches for the use of the Caede notation as a design tool. The process of creating a protocol specification is very similar to creating software designs. The specifier may prefer to use a top-down or a bottom-up approach depending on the degree of development of the protocol. Both approaches are presented to illustrate the flexibility of the notation although only the top-down approach is currently supported by Caede. The support for the bottom-up methodology is left for future work. The section also highlights Caede's ability to provide automated support directly for the graphical notation.

2.2.1) Top Down

The top-down approach involves specifying the highest level (hierarchy) of the Estelle specification first and then successively iterating through structural decomposition until the lowest level has been defined. The pure top-down approach is most useful when presenting or documenting a completed specification. The absence of iteration back to higher levels to correct flaws is a definite shortcoming.

When applied to the alternating bit protocol example, this approach requires the specifier to first select a set of system modules, define the channel types needed for communication between the modules and then connect the modules with channel instances. The modules are initially drawn as labeled black boxes as shown in Figure 2-11. The channel types must then be defined as channel templates (Figure 2-12) and the interface ports must be added to the modules. Channel instances may then be declared between the modules (similar to Figure 2-4).

The internal structure of each module must now be specified as either a set of child modules or a state machine diagram. If the modules are specified previously, as would be the case for the USER and NETWORK modules then the modules may be left blank implying an external definition.

The construction of structure diagrams to represent the parent/child hierarchy of modules involves reiteration though the

module nesting procedure described above. Each module must be internally defined until the modules at the lowest level have been identified.

The state machine diagrams may now be defined. States are added first followed by top's and transitions. The final step is the coupling of transitions to port plugs.

The internal structure of the ALT_BIT_1 module was shown in Figure 2-8. Since the ALT_BIT_1 module has no further nesting of modules, the top-down specification is complete.

2.2.2) Bottom Up

The bottom-up approach is essentially the reverse of the top-down. The lowest levels are specified first and the higher levels are created using the lower levels as building blocks. This approach is most useful during early stages of specification when low level information flows need to be uncovered to ensure modules are adequately supported by channel primitives at higher levels.

As an example of applying the bottom-up approach to the alternating bit protocol, assume that the requirements are modified to include full duplex communication. This means that the alt_bit module must now be capable of both receiving and transmitting messages concurrently and will require separate state machines. The receive (RX) state machine is described below. The transmit (TX) state machine is omitted for brevity, but would be

constructed using the same techniques.

Figure 2-13 shows the RX module state machine. It is initially constructed with states, transitions and couplings to plugs. Note that the interface ports and connecting channels have not yet been addressed. The plugs are created as needed on the boundary of the module. All peer modules must be created in a similar fashion and then added as children to the parent module (ALT_BIT_2). As shown in Figure 2-14, the modules are shown with plugs exposed and without interface ports.

Interface ports are then added to the modules and channel instances are created to connect the modules. The plugs are then repositioned into the appropriate interface ports. Channel templates may then be automatically created by specifying a role for each interface port. The final design for the ALT_BIT_2 module was previously given in Figure 2-5. The channel template for the RX_U channel instance is shown in Figure 2-15.

2.2.3) Automated Support

The ability to deal with large volumes of information becomes an obstacle as the complexity of a specification increases. Caede assists the specifier through automated support in the form of syntax management, views and port plug management. The current state of the support is still experimental and suggestions for enhancements are made in section 5.

Syntax management is provided at all levels of specification. The major aim is to prevent the specifier from drawing diagrams which violate the allowable icon connections presented earlier in this section. Syntax management ensures that the resulting diagrams can be mapped to valid Estelle specifications. This feature is analogous to a language oriented editor for text, except graphical icons are handled instead of text strings.

Graphical syntax management is illustrated during channel connection. The specifier is prevented from connecting a channel between two interface ports having the same role. Creating a channel between two ports with the same role violates the basic Estelle syntax for the "connect" operation and is therefore not permitted graphically.

Views allow the designer to deal with cluttered diagrams by creating a new diagram showing only the information pertinent to a particular aspect of the diagram. Sub-views are created from a master view by filtering out icons which are not relevant to the desired sub-view. Filtering hides the selected icons on the sub-view but does not modify the master view. "Reverse" filtering may also be used to re-display icons which are hidden by previous filtering of a view.

The state machine shown previously in Figure 2-8, contains many event couplings. The view of just the states and transitions gives a much clearer picture of the state machine (Figure 2-7).

The sub-view of Figure 2-7 was created by filtering the master view of Figure 2-8.

The sub-view shown in Figure 2-16 shows all the transitions out of the ESTAB state. This view closely parallels the ability of SDL to show all transitions from a state.

Further automated support provides port plug management between template diagrams and interface ports on modules. As previously defined in section 2.1.3, port plugs are automatically added to an interface port when a channel instance is created. The port plug management automatically guarantees that the port matches the channel type without requiring manual checking.

The port plug management is illustrated between Figures 2-3, 2-4 and 2-5. The template in Figure 2-3 defines the U_ACCESS_POINT channel type and the primitives supported by each role. The U channel on Figure 2-4 is an instance of the channel type. The ALT_BIT_2 module is connected with the provider role. The U_provider external interface port shown on the structure diagram for the ALT_BIT_2 module in Figure 2-5 has the port plugs exposed. The type of plug (solid or hollow) and plug labels have been added automatically by Caede and therefore are guaranteed to match the channel template.

Figure 2-1: Icons for Estelle

Figure 2-2: Channel Template Structure

Figure 2-3: U_ACCESS_POINT Channel Template

SYSTEM

Figure 2-4: Connected Alternating Bit Protocol Modules

Figure 2-5: Internal Structure of ALT_BIT_2 Module

Figure 2-6: Module Structure with State Machine and Nested Modules

Figure 2-7: View of States and Transitions for ALT_BIT_1 State Machine

Figure 2-8: Coupled ALT_BIT_1 State Machine

Figure 2-9: Text Specification Format of Estelle Module

Figure 2-10: Mapping of Caede Diagram Contents to Estelle Text

SYSTEM

Figure 2-11: Alternating Bit Protocol Modules

Figure 2-12: Channel Templates for Alternating Bit Protocol

Figure 2-13: RX Module State Machine

ALT_BIT_2

Figure 2-14: Unconnected ALT_BIT_2 Modules

Figure 2-15: `RX_U_CHNL` Channel Template

Figure 2-16: View of Transitions Out of ESTAB State

3. Case Study : RMS Example

The previous section has introduced the graphical notation for Estelle starting from a pre-existing external specification for the simple alternating bit protocol (Appendix A). This section shows the development of an Estelle specification for the Remote Mailbox System protocol [24] using Caede diagrams as the starting point. The developed protocol specification is then used in Chapter 4 to illustrate the potential for automatic generation of execution code.

The particular case study chosen is the Remote Mailbox System example known as RMS. RMS has been used during the last three years in various courses at Carleton as an illustration of embedded system design and implementation. This example was chosen because it is concrete, complete (having seen several successful implementations) and is well understood by the author not only in protocol terms, but also in implementation terms.

A brief overview of RMS is provided although the intent of the section is to highlight the specification process rather than the details of RMS. A top-down approach is used and the Service layer (similar to the OSI transport layer [28]) is the main focus. Section 3.1 describes a representative sub-set of the Caede diagrams for the specification and section 3.2 presents the actual Estelle specification framework which is generated from the diagrams. The discussion of identified issues is left for Chapter 5.

RMS is a communication service which provides message passing capabilities between applications which may be executing on physically separate nodes of a network. Figure 3-1 shows the major components. User applications executing on a node may send messages to globally named mailboxes. It is the responsibility of the application using RMS to empty the mailboxes as messages are received.

Messages are sent from a source node to the destination node and may be required to pass through intermediate nodes. In Figure 3-1, User A1 sends messages to mailboxes resident on nodes A, B and C. Message MSG1 for Mailbox MA1 never leaves the node, however RMS must still be used to deliver the message. Message MSG2 for Mailbox MB1 is sent directly to node B. Message MSG3 for Mailbox MC1 must first pass through intermediate node B to reach the destination node C. A status response is returned to the User by RMS indicating whether the message was successfully delivered or assumed to be undelivered.

The physical links between nodes are assumed to be unreliable and therefore a simple layered protocol architecture is used to provide reliability. The three layers are:

- 1) Service Layer: provides end-to-end reliability (destination returns an acknowledgement when a message is delivered to a Mailbox).

- 2) Link Layer: provides reliability between physically connected nodes (similar to alternating bit protocol).

3) Driver Layer: ensures data integrity (checksum) and supports hardware I/O during the transmission and reception of messages.

3.1) Caede Diagrams

RMS is shown as a system component for a single node in Figure 3-2. The User module communicates with RMS through two independent channels. RMS_SEND supports sending messages while RMS_RCV allows RMS to deliver messages to Mailboxes located on the node. The communication hardware has been represented as an Estelle module and a channel exists to allow each physically connected link to be handled independently.

Figure 3-3 shows the RMS internal structure. For this example, the number of connected links is arbitrarily set at three. The single SERVICE module supports all User interaction while a unique Link/Driver pair exists for each of the hardware links to adjacent nodes. In this example the SERVICE module is developed and it is assumed that the Link and Driver modules have already been specified. This approach implies that the communication primitives between the User and Link modules have already been defined. Figure 3-4 shows the channel templates for the User and Link interfaces. RMS_SEND and RMS_RCV are instances of U_SEND_CHL and U_RCV_CHL types respectively. S_L_1, S_L_2 and S_L_N are instances of the S_L_CHNL type.

The internal structure of the SERVICE module is shown in

Figure 3-3: RMS Architecture for a Node

The internal state machine for the SEND module is shown in Figure 3-7. The module is READY until it has been allocated to a message. The message is then sent to the link switch for transmission across the appropriate link and the process waits for an acknowledgement. The appropriate status is returned to the User when either an acknowledgement is received or a timeout occurs. The module then returns to the READY state and the SEND_MGR is informed to allow the module to be reallocated.

3.2) Estelle Specification

The generation of the Estelle specification from the Caede diagrams is presented in a similar top-down fashion. The framework of the specification is presented with clear indications of the portions which must be added as textual input. As with the Caede diagrams, only representative portions of the specification have been included.

The highest level specification for the node is the SYSTEM specification corresponding to Figure 3-2. A skeleton of the specification is shown below to show the definition of the U_SEND_CHL type and the RMS module. Although many parts of the specification have been omitted, comments have been included to indicate the parts which would normally be present in all module specifications.

specification SYSTEM;

```

/* specification body */

/* declaration part */

/* constant definitions omitted */

/* type definitions - Pascal types omitted */

/* channel type definitions */

channel U_SEND_CHL ( user, provider) /* from Figure 3-4 */

    by user :

        send_msg (u_msg_rec : u_msg_rec_type);

    by provider :

        stat_resp (status : status_type);

/* module type definitions */

module RMS_type process /* from Figure 3-2 */

    (S_provider : U_SEND_CHL(provider) queue individual;

     R_user : U_RCV_CHL(user) queue individual;

     HW_1_user : HW_CHANNEL(user) queue individual;

     HW_2_user : HW_CHANNEL(user) queue individual;

     HW_N_user : HW_CHANNEL(user) queue individual);

    body RMS_body for RMS_type; /* body omitted */

/* variable declaration part omitted */

/* procedure and function declaration part omitted */

/* initialization part omitted */

/* transition declaration part omitted */

/* termination part omitted */

end. /* end of SYSTEM specification */

(Note: channel primitive parameters, "process" and "queue
individual" added as text.)

```

The RMS module body specification would include the

definition of the channels and modules shown in Figure 3-3. The SERVICE module would be defined as one of the modules. The specification of the internal elements of the SERVICE module (Figure 3-5) would again be nested in the SERVICE body declaration part. The following specification fragments would appear in the SERVICE module body declaration and initialization parts to provide the initialization of the SOURCE module as extracted from Figure 3-5. The initialization requires the appropriate channel and module types to be declared and then instances of the modules are created and bound by instances of channels.

```

/* in declaration part */

/* channel declarations - U_SEND_CHL and U_RCV_CHL previously
   declared at System specification level */
channel SOURCE_CHL( user, provider ); /* for S_SWITCH */

/* module declarations - only SOURCE shown */
module SOURCE_type process
    (S_provider : U_SEND_CHL( provider );
     R_user : U_RCV_CHL( user );
     S_SWITCH_u : SOURCE_CHL( user ); );

/* variable declarations for module instances */
var
    SOURCE : SOURCE_type;

    /* similar for other modules */

/* initialization part */

initialize
begin

```

```

/* initialize all modules */
init SOURCE with SOURCE_body;

/* similar for other modules */

/* bind channels to SOURCE */
/* 1. attach SOURCE to external interaction point */
attach S_provider to SOURCE.S_provider;

/* 2. create R_LOGAL */
connect SOURCE.R_user to U_SWITCH.R_S_prov;

/* 3. create S_SWITCH */
connect SOURCE.S_SWITCH_u to L_SWITCH.S_SWITCH_p;

end; /* end of initialization */

```

The major portion of a specification not yet addressed is the transition part. The SEND module shown in Figure 3-7 is used as an example. A sub-set of the transitions are given below and would be located in the transition part of the body of the SEND_type module.

```

/* transition part of SEND module */

trans. send

from READY /* source state */

when SEND_prov.send_msg(u_msg_rec) /* message ready */
begin /* transition block */

    /* append additional necessary data */
    s_msg_rec := build_s_msg(u_msg_rec);

    /* send the message */
    output S_SWITCH_u.send_msg(s_msg_rec);

end;

```

```

to WAIT_ACK /* destination state 2/

trans valid_ack

from WAIT_ACK

when S_SWITCH_u.got_ack(s_msg_rec) /* ack received */
provided ack_match(s_msg_rec) /* ensure valid ack */
begin

    /* inform user */

    output STAT_prov.stat_resp( s_msg_rec.user_id,
                                delivered );

    /* inform SEND manager - ready for another message */
    output SEND_MGR_u.ready( send_id );

end;

to READY

```

In the transitions shown, build_s_msg and ack_match are functions which must be entered as text. Build_s_msg appends a message identifier and other information needed by the destination node to return an acknowledgement. Ack_match is a boolean function which ensures the acknowledgement received is for the last message sent. The parameters for channel primitives must also be entered as text.

Figure 3-1: RMS for Connected Nodes

Figure 3-2: Architecture for a Single Node

Figure 3-3: RMS Architecture for a Node

Figure 3-4: Channels for SERVICE Interaction Points

Figure 3-5: SERVICE Module Internal Structure

Figure 3-6: SOURCE Module Internal Structure

Figure 3-7: SEND Module State Machine

4. Execution of Estelle Specifications

The direct execution of Estelle specifications is desirable for testing/verification of a protocol as well as direct generation of a target software design. This chapter presents a model for the execution of a protocol independent of its implementation. The creation of an implementation software design directly from the Estelle specification to produce usable target software has not been achieved by the approach of this chapter. The issues involved in producing implementation software designs directly from the specifications are discussed in Chapter 5.

The major issues in executing Estelle specifications are identified in the first section. The execution model presented in the second section provides the foundation for an executable specification. Guidelines for automatic code generation for the execution model are presented in the final section including detailed code to implement a state machine.

4.1) Execution Issues

Three major execution issues must be addressed in the execution of an Estelle specification: The hierarchy execution rule (HER) imposed by the structure of the specification presents a major synchronization problem. (The term "hierarchy execution rule" originates in this report as there is not a standard Estelle term to define the restriction.) The handling of non-determinant choices at both the state and state machine levels introduce further scheduling problems. The unrestricted use of exported

know when transitions are enabled. Transitions will be enabled in a module when messages arrive in connected channels or when the enabling conditions on spontaneous transitions are satisfied. The presence of messages in channels is monitored by the Dispatcher while spontaneous transitions are checked by Workers.

Estelle requires that a state machine must have an enabled transition if any messages are ready in connected channels. If a state machine is in a state where there is a message ready in a channel but no transition is enabled, then a protocol design flaw exists. The Dispatcher task may therefore determine whether a module is ready to make a transition by polling the connected channels. If a message is ready for the module, then the module must be ready to make a transition.

A module may also be ready to make a transition if a spontaneous transition is enabled. The Worker/Dispatcher model restricts variables exported by one module from being used in the enabling conditions of other modules. The spontaneous enabling conditions may therefore be evaluated when a state is entered and the conditions are guaranteed not to change until a transition is executed. The guarantee is possible since only local variables may be used in a spontaneous enabling condition and the variables may only be modified by the module during a transition.

Detection of enabled spontaneous transitions is the responsibility of the Worker tasks. After executing a transition

structured into nested processes. In the example of Figure 4-1 the maximum possible concurrency of modules executing transitions simultaneously is six. (The leaf processes are C, D, E, F, H and I).

4.1.2) Non-Determinant Choices

Non-determinant choices are inherent in Estelle's behavioural model. Estelle leaves the resolution of non-determinant choices as an implementation decision with the restriction that the specification may not be dependent on the resolution scheme.

The most obvious case of non-determinism arises when more than one of the transitions from a state are enabled. This could quite easily occur if a module is connected to more than one channel and messages arrive from connected modules while the module is blocked by the HER. The resolution of this situation is aided by the optional specification of priorities for transitions. A tie in priority still presents a non-determinant choice. This type of non-determinant situation is found frequently in languages equipped with inter-process communication mechanisms. In Ada it is possible that two rendezvous may be ready when a selective accept is issued. Ada resolves this situation by choosing one of the rendezvous at random.

A second type of non-determinant situation is slightly more subtle and is caused by the HER. It is possible that a module and one of its ancestors or descendants may both be ready to execute

transitions simultaneously. A non-determinant choice exists as to the selection of which module may proceed to execute. The other module will remain blocked due to the HER. The Estelle language does not contain a priority mechanism to allow the specifier to exercise some control over the resolution of this form of non-determinism. Other multitasking environments like Ada provide task priorities to resolve this situation. It must be noted that it is still possible to have ties in the Ada priorities similar to the Estelle module non-determinism. Ada resolves these ties by random selection.

4.1.3) Exported Variables

The exporting of variables becomes an execution issue only when variables exported from one module are used in transition enabling conditions ("provided" clauses) of another module. This use of exported variables is undesirable since modifications to the exported variable must be monitored continuously by all modules concerned. This type of control coupling is restricted in other languages and blocking conditions are evaluated only when inter-process communication occurs.

4.2) Dispatcher/Worker Model

o The Dispatcher/Worker model provides an executable structure for an Estelle specification based on Ada constructs. The architecture of the model is shown in Figure 4-2 and the discussion of the model is based on the diagram. The design of the model is not immediately apparent when compared to the

components in an Estelle specification. This section presents the elements in the model while the reasons for the resulting modularity is explained in Chapter 5. A very brief explanation of the icons in the diagram is given before discussing the model. A more detailed description of the graphical notation of Ada may be found in [12,2].

Tasks are represented by parallelograms (eg. Dispatcher task). A rendezvous is represented by an arrow originating on the boundary of a task and terminating on a rendezvous entry nested in the accepting task. The rendezvous entry is represented by a trapezoid positioned with one side superimposed on the accepting task. An example of a rendezvous is shown by the Worker task rendezvousing with the GET_WORK entry on the Dispatcher task. Packages are represented by boxes. Interface procedures are shown as boxes nested in a module with one side superimposed on the enclosing package. The Module packages have INIT and EXECUTE interface procedures. Interface procedures may be called in a fashion similar to rendezvous entries.

The model centers around the Dispatcher task which manages Workers from the Worker task pool. The Workers are dynamically assigned to modules as transitions are identified as being enabled. The model uses tasks only to provide concurrency when possible. The Estelle state machines are implemented as passive procedural packages. Bidirectional channels are implemented by Channel packages. The Channel packages include a pair of queues.

4.2.1) Dispatcher Task

The Dispatcher task controls execution of the model. The major function performed by the Dispatcher is assigning Worker tasks to execute transitions. The assignment of Workers requires: determining when a module is ready to make a transition, resolving non-determinant choices between all modules ready to make transitions, and enforcing the hierarchy execution rule. Although this initially appears as a large responsibility, the actual implementation of the functionality will be shown to be straightforward.

The primary function of the Dispatcher task is the assignment of Worker tasks. The Workers wait at the GET_WORK entry until assigned to a module. An assigned Worker reports back to the WORK_DONE entry to inform the Dispatcher that the transition is complete. The Worker then returns to the GET_WORK entry for a new assignment.

The Dispatcher maintains a list of processes ready to make transitions and the work assignments are drawn from the "ready list". Once a module is detected as being ready it remains in the ready list until a Worker is assigned to execute a transition. The module is guaranteed to remain ready until executed since the execution of a transition is the only way in which an enabling condition can be cleared.

The management of the ready list requires the Dispatcher to

know when transitions are enabled. Transitions will be enabled in a module when messages arrive in connected channels or when the enabling conditions on spontaneous transitions are satisfied. The presence of messages in channels is monitored by the Dispatcher while spontaneous transitions are checked by Workers.

Estelle requires that a state machine must have an enabled transition if any messages are ready in connected channels. If a state machine is in a state where there is a message ready in a channel but no transition is enabled, then a protocol design flaw exists. The Dispatcher task may therefore determine whether a module is ready to make a transition by polling the connected channels. If a message is ready for the module, then the module must be ready to make a transition.

A module may also be ready to make a transition if a spontaneous transition is enabled. The Worker/Dispatcher model restricts variables exported by one module from being used in the enabling conditions of other modules. The spontaneous enabling conditions may therefore be evaluated when a state is entered and the conditions are guaranteed not to change until a transition is executed. The guarantee is possible since only local variables may be used in a spontaneous enabling condition and the variables may only be modified by the module during a transition.

Detection of enabled spontaneous transitions is the responsibility of the Worker tasks. After executing a transition

the spontaneous transitions for the new state are checked. If a spontaneous transition is enabled then the Worker relays the ready status to the Dispatcher when making the WORK_DONE rendezvous.

The assignment of work from the ready list will involve non-determinant choices when more than one module is ready to execute simultaneously. The recommended resolution scheme is random selection. The important point is that the resolution of this type of non-determinism is handled entirely by the Dispatcher and any modifications to the scheme will influence only the Dispatcher. It is also important to note that the Worker tasks may not continue executing a process if a spontaneous transition is enabled when a new state is entered. The Worker must report the ready status to the Dispatcher to allow the Dispatcher to resolve any non-determinant choices before re-assigning the module for execution.

The HER requires special consideration when assigning work from the ready list. Once a Worker has been assigned to a module, the HER prevents ancestors and descendants from executing until the module completes the transition. The influence of HER blocking is managed by changing the status of affected modules in the ready list from ready to ready-blocked. The status of blocked modules is returned to ready when the Worker reports back to the Dispatcher indicating that the transition causing the blocking is complete. When a Worker reports that a transition is complete, the Dispatcher must only check the channels of modules which are unblocked by the HER and are not already in the ready list.

The management of the HER requires a Dispatcher database. A hierarchy tree describing module nesting structure must be available to allow the Dispatcher to determine which modules are related by the HER. The Dispatcher needs to know which channels are connected to each module and uses this information when determining which modules have messages waiting. The Dispatcher database information can be generated directly from the Caede structure diagrams.

4.2.2) Worker Tasks

The Worker tasks are responsible for executing transitions and evaluating spontaneous transitions. The Workers receive assignments at the GET_WORK of the Dispatcher task. The Worker examines the enabling conditions for all possible transitions out of the current state of the assigned module and if no transition is enabled then a protocol error is assumed to have occurred. If more than one transition is enabled then the Worker must resolve the non-determinant situation. As with the Dispatcher task, the recommended resolution scheme is random selection. As described previously, after executing a transition the Worker must evaluate the spontaneous transitions for the newly entered state and report any enabled conditions to the Dispatcher.

4.2.3) Channels

Estelle defines channels as being supported by infinite queue. All messages which are initiated through channel primitives

are queued until accepted by the connected module. The Channel packages in the Dispatcher/Worker model implement the queues as linked lists. A FIFO scheme is used but if an alternate scheme is desired, the necessary changes are local to the Channel package type only. Five interface procedures allow initialization, sending, receiving and polling. Polling examines the entry at the head of a channel queue and is used when evaluating enabling conditions. The entry at the head of the queue will not actually be removed until a transition is executed which consumes (receives) the entry.

4.2.4) Modules

Modules are implemented as passive Module packages. The packages have interface procedures to allow initialization and execution. The execution entry is called by a Worker to execute a transition and all processing is internal. The packages require a local data area to maintain the state and operational variables. A detailed example of a Module package is given later.

4.3) Automatic Code Generation for Caede

Ada code for the Dispatcher/Worker model can be automatically generated from the Caede representation of an Estelle specification. Although Caede does not currently support this ability the following guidelines show that automatic code generation is feasible.

4.3.1) Dispatcher Task

The Dispatcher task executes the following algorithm:

- 1) Initialize all components in the model. Initialization is done at two conceptual levels. The first is the initialization of the Ada structures and the second is the Estelle specification module initialization parts. The initialization of the components should be followed in the exact order indicated in the Estelle specification.
- 2) Check all module connections and build the initial ready list.
Loop forever:
 - 3) Dispatch as many concurrent Workers as possible under the restrictions of the HER:
While any module is ready:
 - 3.1) Select a module from the ready list (non-determinant choice).
 - 3.2) Change status of all modules which will be blocked by the HER.
 - 3.3) Dispatch a Worker to the selected module.End while;
- 4) Wait for WORK_DONE and associated module status.
- 5) If module has a spontaneous transition ready, then add module to the ready list.
- 6) Unblock all modules previously influenced by the module under the HER. Previously ready modules now unblocked will already be in the ready list.
- 7) Check channel connections to all modules unblocked and

previously not ready and add to ready list if appropriate.

End loop forever;

The automatic generation of code for a Dispatcher task depends on the Estelle specification only for the construction of the hierarchy tree and lists defining the connection of channels to the modules. The database constructs generated for a specification do not influence the actual Dispatcher code which processes the constructs. The Dispatcher database can be generated automatically from Caede structure diagrams. The table/list driven design allows the Dispatcher code to be defined by a general Dispatcher task template:

4.3.2) Worker Tasks

The individual Workers are all identical and do not contain any Estelle specification information. The Worker task pool may therefore be created as instances of a single task type. Estelle related execution is carried out by the Workers in the procedures of the assigned Module package. The Worker tasks implement the following algorithm:

Loop forever:

- 1) Wait for work at GET_WORK.
- 2) EXECUTE a transition for the assigned module.
- 3) Report the new status of the module at WORK_DONE.

End loop forever;

4.3.3) Channels

The execution of a transition may involve interactions with a channel as indicated by the coupling of state machines to interface ports. The Channel packages have the internal architecture shown in Figure 4-3. An individual Queue package exists for each role and the "selector" interface parameter determines the queue to be accessed. The User Queue RECEIVES messages from the module connected with the Provider role and SENDS to the module connected as the User role. The MUTEX tasks exist to provide mutual exclusion for the possible situation where the User and Provider modules attempt simultaneous access to a queue. ~~The~~ POLL interface accesses the queues directly.

The Channel package interfaces are complicated by the variety of information which may be queued. Each unique channel primitive may have a unique number and type of parameters. When a Channel is polled, the caller does not know in advance what information will be returned. The uncertain parameter format is difficult to implement in Ada due to the requirement that all interface parameters be defined during compilation.

The proposed solution is to split the polling into two parts and to overload an interface procedure. The first part of the polling sequence is to call the single POLL_PRIMITIVE function which returns a boolean indicating whether the queue has any data and the identifier of the primitive associated with the message on the top of the queue. If it is necessary to look at the message contents, the caller then uses the primitive identifier to select

a call with appropriate parameters to POLL_MESSAGE. The POLL_MESSAGE procedure is overloaded with a different procedure for each primitive supported by the channel. The Channel interface procedures to SEND and RECEIVE are overloaded in an identical fashion to POLL_MESSAGE.

The unique interface procedures for each Channel package require individual packages to be constructed. The automatic generation of the packages can still be accomplished from the Caede specification based on the set of primitives (with associated parameters) supported by each channel template. An overloaded procedure must be created for each role/primitive combination. The internal operation of each overloaded procedure differs only in the particular data parameters handled.

An issue which has not been addressed is how to queue the various data formats in a single queue. The proposed structure allows the Estelle module state machines to function regardless of the queuing method. Furthermore, the implementation of the queuing method is not influenced by the Estelle specification since the issue is a general Ada programming problem and is not Estelle dependent.

One possible solution to the queuing problem would be to maintain an individual queue for each primitive and a master queue for the primitive identifiers. The individual queues would then deal with fixed parameter formats. The order of the

primitive identifiers in the master queue would direct the removal of messages from the individual queues.

The Queue packages are entirely passive. Although both the Worker and Dispatcher tasks may require simultaneous access to the POLL procedure, simultaneous polling is not a problem since no data is modified. Simultaneous POLLing and RECEIVEing will never occur since only the assigned Worker may execute a RECEIVE and the Dispatcher will never POLL a channel role while the connected module is active. The internal architecture of the Queue packages will depend on the resolution of the previously raised queuing issue.

4.3.4) Modules

The Module packages contain Estelle execution information. The implementation structure for a Module package is shown in Figure 4-4. New software icons have been introduced for data (octagon, eg. STATE) and internal procedures (box with an extended side, eg. EVAL_SPONT).

The package structure closely follows the Estelle transition model described in section 2-1. Each transition has an enabling condition function which returns a boolean indicating whether the transition is enabled and a transition block procedure to perform the actual execution of the transition.

The module database contains all required state information

The database also includes a module transition list defining all transition records for the module. A transition record is defined for each transition and contains the priority, next state and enabled status.

The enabling condition functions, transition block procedures and the module transition list can be automatically generated from the Caede diagrams and accompanying text.

The algorithm for executing a transition is given below:

- 1) Select an enabled transition.
 - 1.1) Evaluate all enabling conditions for transitions out of the current state.
 - 1.2) Resolve any non-determinant choices.
- 2) Execute the selected transition.
- 3) Evaluate all spontaneous transition enabling conditions for the state entered and return the ready status if a transition is enabled.

The EXECUTE procedure directs the execution of a transition. The enabled status of each transition record is initially reset to "disabled". EVAL_SPONT and EVAL_INPUT are then called to evaluate the enabling conditions for all transitions out of the current state. EVAL_SPONT and EVAL_INPUT call the enabling function for each spontaneous and input driven transition respectively. The returned status from each enabling function is saved in the enabled status of the appropriate transition record. EVAL_SPONT

and EVAL_INPUT both return a status value indicating whether any transition is enabled. If no transition is enabled, then an error condition is assumed.

EXECUTE calls the RESOLVE procedure to resolve any non-determinant choices. RESOLVE uses the enabled status in the module transition list to determine which transitions are enabled. The identifier for the transition to be executed is returned. The appropriate transition block procedure is then called through the DO_TRANS procedure to perform the actions associated with the transition.

After executing the transition, the state variable is updated to the next state specified in the transition record and then EVAL_SPONT is called to determine whether any spontaneous transitions are enabled for the new state. The status returned from EVAL_SPONT is returned by the Worker to the Dispatcher task during the WORK_DONE rendezvous.

4.3.5) State Machines

The following example contains illustrative fragments of skeleton code for the RMS SERVICE SOURCE SEND module state machine (Figure 3-7). The example requires the SEND_Q_CHL channel package specification to define the primitives present in the channel. The SEND package Ada specification defining the visibility of the package is presented and discussed before the body is presented.

The following declaration appears in the visible header of

the SEND_Q_CHL package and must be "used" by the SEND package. The type may be automatically derived from the channel template and consists of the enumeration of all primitive supported:

```
type SEND_Q_CHL_primitive_type is (send_msg);
```

The visible portion of the SEND package specification must contain the declaration of the INITIALIZE and EXECUTE interfaces. The interface procedures are common to all Module packages. In this example, the INITIALIZE procedure includes an identification parameter which is used to support the implementation of the SEND process pool. The parameter is used by the SEND module to identify itself and the instances of the connected channels. The following Ada specification is generated:

```
package SEND_state_machine is
    procedure INITIALIZE( SEND_process_id : Send_process_id_type);
    function EXECUTE return BOOLEAN;
end SEND_state_machine;
```

Only a representative portion of the package body has been included. The declaration portion is presented first followed by the procedures and functions common to all modules. The procedures and functions unique to the SEND package are then presented. For automatic generation of code, the SEND package name is prefixed to provide unique names for constants, types and variables.

Declaration part:

```
package body SEND_state_machine is
```

```

-- constants
number_of_transitions : constant := 4;

-- common to all Module packages
enabled : constant := true;
disabled : constant := false;
no_priority : constant := -1

-- declare state and transition types
type SEND_state_type is (READY, WAIT_ACK);

type SEND_transition_type is
    (tx_msg, ghost, valid_ack, timeout);

-- declare transition record type
type SEND_transition_record_type is
    record
        priority : integer;
        next_state : SEND_state_type;
        status : boolean;
    end record;

-- define array of transition records for state machine
type SEND_transition_array_type is
    array (SEND_transition_type) of SEND_transition_record_type :=
        ( tx_msg => ( no_priority, wait_ack, disabled ),
          ghost => ( no_priority, wait_ack, disabled ),
          valid_ack => ( no_priority, ready, disabled ),
          timeout => ( no_priority, ready, disabled ) );

-- define the transition list
type SEND_transition_list_type is
    record
        number : constant := number_of_transitions;
        list : SEND_transition_array_type;
    end record;

-- declare variables
SEND_state : SEND_state_type;
SEND_transitions : SEND_transition_list_type;

s_msg_rec : s_msg_rec_type;

```

The number_of_transitions is obtained from the Caede diagram as is the enumeration of SEND_state_type and SEND_transition_type. The SEND_transition_record_type is similar to all process transition_record_types and can be created from a general template by inserting SEND_state_type for the next_state type. The

SEND_transition_array_type structure is common to all processes. A unique record is defined for each transition and the next_state is identified directly from the Caede diagram. All transitions are initially disabled and the priority (if present) must be obtained from input text. The SEND_transition_list_type is automatically generated from a template by inserting the SEND_transition_array_type. The state and transition list variables are required for all processes. The s_msg_rec variable is needed to save information used during acknowledgement validation.

Common procedures and functions:

```
-- *****
procedure INITIALIZE(SEND_process_id : in SEND_process_id_type) is
begin
    -- initialize the state
    SEND_state := READY; -- from module initialization

    -- other initialization omitted
end INITIALIZE;

-- *****
function EXECUTE return boolean is
    -- structure common to all modules with details generated from
    -- the diagrams and text

    TRANS_TO_DO : SEND_transition_type;
begin
    -- clear all transition enabled status'
    for I in SEND_transition_type loop
        SEND_transitions.list(I).status := disabled;
    end loop;

    -- evaluate all transition enabling conditions
    IF (EVAL_SPONT or EVAL_INPUT) /= enabled then error;

    -- determine the transition to be done
    TRANS_TO_DO := RESOLVE;
```

```

-- do the transition
DO_TRANS( TRANS_TO_DO );

-- assign new state value
SEND_state := SEND_transitions.list(TRANS_TO_DO).next_state

-- check if any spontaneous transitions enabled and
-- return status
return EVAL_SPONT;

end EXECUTE;

-- *****
function EVAL_INPUT return boolean is
-- structure common to all modules with details generated from
-- the diagrams and text

result : boolean;

begin
-- initialize return value
result := disabled;

-- use current state to decide which transitions to check
case SEND_state is
when READY => -- check only tx_msg
    SEND_transitions.list(tx_msg).status := eval_tx_msg;
    result :=
        result or SEND_transitions.list(tx_msg).status;

when WAIT_ACK => -- check ghost, valid_ack
    SEND_transitions.list(ghost).status := eval_ghost;
    result := result or SEND_transitions.list(ghost).status;

    SEND_transitions.list(valid_ack).status :=
        eval_valid_ack;
    result :=
        result or SEND_transitions.list(tx_msg).status;

end case;

return result;

end EVAL_INPUT;

-- *****
procedure DO_TRANS( TRANS_id : SEND_transition_type );

-- structure common to all modules

begin
-- case statement based on transition selects procedure
case TRANS_id is

```

2

OF/DE

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS,
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

```

    when tx_msg => tx_msg_proc;
    when ghost => ghost_proc;
    when valid_ack => valid_ack_proc;
    when timeout => timeout_proc;
end case;

end DO_TRANS;

```

All modules contain the INITIALIZE, EXECUTE, EVAL_SPONT, EVAL_INPUT, RESOLVE, and DO_TRANS procedures and functions. The basic code structure for each procedure/function is common to all module state machines while the details are unique for each module. The EVAL_SPONT function has been omitted due to its similarity to EVAL_INPUT. RESOLVE has also been omitted.

In both EVAL_INPUT and DO_TRANS, case statements are performed to determine which module dependent functions or procedures should be called. For EVAL_INPUT, the enabling function names have been created by prefixing "eval_" to the transition names. The actual functions to be called are obtained directly from the Caede diagrams as the input driven transitions originating from each state. DO_TRANS selects a transition block procedure. The name of the procedure is created by appending "_proc" to the transition name.

Procedures and functions unique to SEND:

```

-- *****
function eval_tx_msg return boolean is
primitive_id : SEND_q_chl_primitive_type;

begin
    -- poll channel for a message
    if
        SEND_Q_CHL(SEND_process_id)

```

```

        .POLL_primitive( provider, primitive_id)
        then return enabled;
        else return disabled;
    end if;

end eval_tx_msg;

-- *****
procedure tx_msg_proc is
    u_msg_rec : u_msg_rec_type;

begin
    -- get message
    SEND_Q_CHL(SEND_process_id).
        RECEIVE( provider, send_msg, u_msg_rec );

    -- build service message
    s_msg_rec := build_s_msg(u_msg_rec);

    -- send the message
    S_SWITCH_C(SEND_process_id).
        SEND(user, send_msg, s_msg_rec );

end tx_msg_proc;

end SEND_state_machine;

```

The module dependent portions of the state machine code are the enabling functions, transition block procedures and other support procedures and functions. The majority of the information for the components is obtained from textual input. The input driven enabling functions will at least poll a channel and may require addition "provided" processing. The transition block procedures must actually read any required messages from the channel, process data as needed and send messages to channels. The enabling function and transition block procedure for the tx_msg transition have been included. Tx_msg_proc calls the build_s_msg support function as an exact parallel to the Estelle specification as shown in Figure 3.2

Figure 4-1: Example of Process Hierarchy in a Specification

Figure 4-2: Architecture of the Dispatcher/Worker Model

Figure 4-3: Internal Structure of a Channel Package

Figure 4-4: Internal Structure of a Module Package

5. Discussion

This chapter discusses issues and implications which arise from the material presented in the previous sections. The first section discusses Caede's graphical notation for Estelle. Specific enhancements are identified to develop the initial "proof-of-concept" notation to a more complete Estelle representation. Section 5.2 examines general issues which a graphical notation must address in the specification of protocols. Section 5.3 discusses the implications of executing Estelle specifications. An architecture with Ada tasks mapped one-to-one to Estelle processes is used as a starting point. It is shown how the Dispatcher/Worker model evolved from this initial design. Section 5.4 addresses the original goal of complete alignment of the protocol specification process with software engineering. The final section presents some recommended enhancements to Caede as a design tool. These enhancements do not deal with representing Estelle but are concerned with the manipulation of icons through Caede's human interface.

5.1) Enhancements to the Graphical Notation for Estelle

This section identifies areas which require work to advance the current notation to a complete graphical representation for Estelle. It should be noted that the evolving nature of the Estelle language may require future graphical enhancements. The current graphical representation of module sub-structuring is ambiguous in certain situations and should be revised. The current shortcoming is discussed in section 5.1.1 while actual

revision is left for future work. Dynamic operations on objects is the only area outstanding to complete the representation of Estelle. Some suggestions are made in section 5.1.2 regarding the integration of dynamic operations but detailed implementation is left for future work. The graphical differentiation between process and activity modules is discussed in section 5.1.3. The final section summarizes the information which is left for input as text.

5.1.1) Representation of Module Sub-structure

This section discusses the implications of using the "_FSM" naming convention to distinguish parent state machines from nested modules. The graphical inconsistency was initially identified in section 2.1. The current use of the naming convention is undesirable since it complicates the interpretation of structure diagrams.

The graphical notation for module structure represents child modules as black boxes nested inside the parent module's boundary. This is attractive since all communication with the modules may be represented through channels. The current notation also represents the parent module's state machine as a nested module (black box). The only graphical distinction between the parent state machine and other nested modules is the use of the "_FSM" naming convention to identify the state machine. The notation was originally proposed to allow interactions between parent and child modules to be represented consistently by channel connections.

The notation places external interface ports on the boundary of the parent module while internal interface ports are drawn on the boundary of "_FSM" boxes. This appears to indicate that the internal interface ports belong to the "_FSM" module when in fact both the external and internal interface ports are part of the parent module.

The notation introduces inconsistency in the mapping of the graphical structure diagrams to the structure of an Estelle module specification (Figure 2-9). Inconsistency is introduced in the two ways described below.

The parent state machine is defined in the transition part of the specification while nested modules are defined in the declaration part. The graphical notation is inconsistent since the nested black box notation suggests the information associated with the "_FSM" icon should be in the declaration part instead of the module transition part.

All parent module interaction points are defined in the header part of the module specification. The child module interaction points are defined in the declaration part of the parent module. The notation is again ambiguous and suggests that the interface ports on the "_FSM" icon should be specified in the declaration part instead of the header part.

Although a situation to illustrate the inconsistency did not

arise in the RMS Service layer, the problem did arise in the specification of the RMS Link layer [27]. Figure 5-1 shows the structure diagram for an RMS LINK module. The LINK_1 module is the parent module and the S_SWITCH, D_SWITCH, LINK_RX, LINK_TX and LINK_TX_Q modules are children of LINK_1. The process state machine for the LINK_1 module is indicated by the LINK_1_FSM icon. The state machine does not outwardly appear any different from the nested modules and the difference is implied through the "_FSM" naming convention.

Further ambiguity is illustrated by the external interface ports (S_L_1_prov and L_D_1_user) and the LINK_1_FSM interface ports. The interface ports are all specified in the LINK_1 module header even though they appear to be on separate modules. The current notation implies that the external interfaces are specified at the same level as the interfaces to the "_FSM" module.

The use of the naming convention to imply structural information is less than ideal since it complicates the interpretation of the diagrams. A more optimal solution is left for future work.

5.1.2). Dynamic Operations on Objects

The notation for Estelle must be enhanced to represent dynamic operations on objects. Dynamic handling of objects will introduce new relationships between structure and state machine

diagrams. Additional icons will be needed to represent both the dynamic operations and the dynamic access to the initialization and termination parts of a module. The graphical representation of dynamic operations is left for future work.

Dynamic operations will introduce relationships which logically bridge structure and state machine diagrams. The bridging is necessary since the dynamic manipulation will occur during state machine transition blocks while the objects being manipulated are located on structure diagrams.

A suggested approach to bridging the diagrams is to use logical connection points on the edge of module boundaries. On structure diagrams, the logical connection points would be connected to the objects being manipulated by icons representing dynamic handling. On the state machine diagrams, the logical connection points would be connected to top's on the transitions which perform the dynamic operations.

New icons may be needed to identify the operations of "init", "connect", "attach", "disconnect", "detach" and "release". The "init" and "connect" operations are currently implied graphically by the existence of state machines and channel instances. The static "attach" operation is currently represented by the dotted arrow (Figure 2-1f) but has not been considered dynamically. The "disconnect", "detach" and "release" operations have not been considered.

Associated with the dynamic management of structure is the introduction of the "terminate" part of the module body and more complicated use of the "initialize" part. Additional enhancements to the graphical notation will be needed to indicate access to these parts during process execution.

5.1.3) Process and Activity Modules

The current notation does not distinguish between process and activity modules. The current solution is to assume a module is a process unless specifically defined as an activity using textual annotation. Although the need to differentiate between processes and activities did not arise in the case study work, a new activity module icon is recommended.

Process and activity modules differ in execution behaviour and sub-structuring rules. Peer processes may execute transitions simultaneously. Peer activities are restricted such that only one of the activities may execute a transition at a time. The activity behaviour guarantees that mutual exclusion will be provided for access to variables shared between activities. Processes may be sub-structured into processes or activities but not both. In contrast, activities may not be sub-structured and may only contain a state machine.

The difference between processes and activities warrants recognition graphically. A suggested solution is to represent activity modules with an "A" in the lower left corner of the

module icon. Absence of the "A" would imply a process module. This solution is similar to the "M" used to distinguish a procedure from a task mainline in the Caede notation for Ada [2].

5.1.4) Textual Information

The representation of parameters describing Estelle objects and the structure of abstract data types are deemed to be unsuitable to graphical representation. These components must be input as text and the human interface considerations for textual input is left for future work. Program action information is also currently input as text but this may change with future developments in Caede.

The following Estelle descriptive parameters are not suited to graphical representation: external channel queue mechanism (common or individual), channel primitive parameter lists, transition priority and transition delay clauses.

Abstract data types (structure and allowable operations) are defined in Estelle using program code. The structure of abstract data types is best represented through textual definitions. Representation of operations on abstract data types depends on the ability to represent action information graphically.

Action information is the detailed operation of procedures and functions and has not been dealt with by the graphical notation. The graphical representation of action information is a

fundamental issue in Caede and is left for future work. The development of a notation for Estelle action information should be postponed until the more general issue of action information in Caede is addressed. The Estelle notation should be consistent with any notation developed for Caede software designs.

5.2) Broader Graphical Issues

The application of the notation to case studies has provided some insight into the more general issue of representing protocols. Bochmann [15] and others [14,29] have attempted to identify relationships and situations which occur frequently in the specification of protocols. This section presents three areas where graphical representation using the notation raises issues. The first is hierarchical dependencies [15] which involve the nesting of a state machine in a state. The second area is the representation of direct coupling [15] which is obscured by the use of abstraction. The third area is the occurrence of switch modules which are again caused by the use of abstraction. Improved handling of the areas is left for future work.

5.2.1) Hierarchical Dependency

Hierarchical dependencies arise when the existence and operation of a process (or processes) depends on another process being in a particular state. Estelle contains no provision for directly specifying modules nested in a state of the parent

process. Given that Estelle is incapable of consistently representing hierarchical dependency it is not surprising that the graphical notation has associated problems.

A typical example of hierarchical dependency occurs in protocol specification when the exchange of messages is allowed only when a communication link is in the "UP" state.

Figure 5-1 illustrates the occurrence of the example situation in the Caede diagrams which resulted from specifying the RMS Link layer. A hierarchical dependence exists between the LINK_1_FSM and the LINK_RX and LINK_TX state machines. The LINK_1 module state machine (LINK_1_FSM) controls the state of the logical link between two nodes. The LINK_RX and LINK_TX processes handle the transmission and reception of messages and may only execute when LINK_1 is in the UP state. The LINK_RX and LINK_TX state machines are nested in the UP state of the LINK_1 state machine and a hierarchical dependence exists.

The representation of the hierarchical dependence is not obvious using the current notation and results in Figure 5-1 being structurally cluttered. The dependency has required the introduction of the "_CNTL" (control) channels between the LINK_1_FSM and all other modules to inform the other modules when LINK_1_FSM enters or leaves the UP state. The presence of control channels between modules directly involved with the hierarchical dependency may be acceptable, however additional channels are

required for the `_SWITCH` and `_QUEUE` modules to control the flow of messages to the `LINK_RX` and `LINK_TX` modules.

It is hoped that future versions of Estelle and the Caede notation will allow a better representation of hierarchical dependency. The graphical congestion may be reduced when dynamic object handling is integrated into Caede. The `LINK_1` module may then dynamically create and connect the `LINK_RX` and `LINK_TX` modules when appropriate.

5.2.2) Direct Coupling and Abstraction

Direct coupling of state machines represents a synchronization process similar to a rendezvous. Direct coupling may lead to potential deadlock situations and therefore must be identified and understood while constructing a specification. A directly coupled relationship is not obvious from the Caede structure diagrams since abstraction prevents showing two state machines and couplings simultaneously. The specification of direct coupling must be addressed in future versions of Estelle.

Direct coupling is present in the interactions between the `SEND`, `SEND_MGR` and `S_QUEUE` processes of Figure 3-6. A simplified version of the coupling is shown in Figure 5-2. The diagram shows the internals of both the modified `SEND` and `S_QUEUE` processes simultaneously. `SEND_MGR` has been eliminated from the example. Note that it is not possible to construct this type of diagram using Caede and it is included for discussion purposes only.

The S_QUEUE process buffers the incoming stream of messages from the send_msg plug on the SEND-prov interface port. The messages are then passed to the SEND process only when SEND is in the READY state.

Direct coupling between the SEND.READY state and the S_QUEUE.MSG_READY state may be illustrated by the following sequence. Assume that the S_QUEUE process is in the EMPTY state and the SEND process is in the SEND_MSG state. If a new message arrives at S_QUEUE then S_QUEUE enters the MSG_READY state. S_QUEUE can not leave the MSG_READY state until SEND enters the READY state. Once SEND enters the READY state, both processes are synchronized. S_QUEUE will dequeue a message and both processes will execute transitions. The two states have been shown to be directly coupled in this situation.

The rendezvous is used to implement synchronization in Ada, however Estelle and the graphical notation do not currently contain methods to identify directly coupled relationships. The Estelle language is still evolving and the existence of an Estelle rendezvous has been identified for future development. If the eventual Estelle rendezvous is similar to an Ada rendezvous then the specification of direct coupling will be possible.

The development of a graphical representation of direct coupling must be left until Estelle addresses the issue.

5.2.3) Switch Modules and Abstraction

"Switch" modules are a direct result of abstraction. The function of the modules is simple, but the required specification is tedious. The high frequency of occurrence of switch modules suggests that a more compact specification method may lead to more compact and unambiguous specifications.

Abstraction hides the implementation details of modules. This is desirable because it allows the communication between modules to be based on information requirements rather than an implementation architecture. Abstraction encourages the number of channels to be minimized and the interfaces to be kept simple.

Switch modules perform a multiplexing function to allow external channel interactions of a parent module to be distributed among internally nested child modules. The direct sharing of parent interaction points by child modules is not allowed in Estelle and therefore the switch modules are required. The switch modules redirect received messages to other child modules. The initiation of primitives at the external interaction point is also done by the switch module on the behalf of the other child modules.

The U_SWITCH module shown in Figure 3-5 allows both the RECEIVE and SOURCE modules to initiate the "deliver" primitive at the SERVICE.R_user interface port. The L_SWITCH module in the

same diagram is more complicated and allows messages to be sent and received between the SOURCE (and RECEIVE) module and any of the connected LINK modules.

The use of switch modules is both tedious and ambiguous. Switch module specification is tedious using the graphical notation since a transition and event coupling must be drawn for each interaction which must be redirected. Switch modules are ambiguous by adding to the structural clutter of a diagram without really contributing to the behaviour of the protocol.

A recommended future enhancement of Estelle (and the graphical notation) is a more compact and less ambiguous representation of the switch module function. This may require re-examining the restrictions on "attaching" parent and child module interaction points.

5.3) Execution Models for Estelle

This section describes the issues associated with the execution of Estelle specifications. The task-per-process model is presented as an initial execution model. The discussion highlights how the execution issues lead to the transformation of the task-per-process model into the Dispatcher/Worker model (presented in Chapter 4). The modifications to the task-per-process model are a consequence of issues primarily associated with channel implementation. The section also includes a brief discussion on the justification of tasks in the execution model.

The task-per-process model is introduced through the simple example in Figure 5-3. The figure forms the basis for all discussion of the model. The components in the figure have been arbitrarily specified to include four processes (PROCESS_S, PROCESS_A, PROCESS_B and PROCESS_C) and two channels (CH_1 and CH_2): Figure 5-3a) shows the specified hierarchy of the processes. PROCESS_S represents the "specification" (highest level module). PROCESS_A and PROCESS_B are peer modules nested in PROCESS_S while PROCESS_C is nested in PROCESS_B.

5.3.1) Task-Per-Process: Version 1

The task-per-process model takes advantage of the natural specification of concurrency in Estelle and Ada to align a unique Ada task per Estelle process. The master control task (MC_TASK) manages the hierarchy execution rule (HER). Channels connecting the processes are shown by the dotted double arrows (CH_1 and CH_2). The channel implementation shown involves direct rendezvous to initiate a primitive.

The MC_TASK must be present to manage the HER in a fashion similar to the Dispatcher task presented in Chapter 4. The HER requires all transitions to be synchronized such that neither the ancestors or descendants of a process begin a transition while the process is executing a transition. Unless execution behaviour to accommodate the HER is built into the underlying multitasking kernel, then all process tasks must be synchronized to prevent HER violations.

The resolution of non-determinant choices and restrictions on use of exported variables must be handled in a similar manner to the Dispatcher/Worker model. The process tasks resolve choices among enabled transitions while the MC_TASK resolves choices among enabled processes. Exported variables must not be used in the enabling functions of spontaneous transitions. Violation of this rule would require "busy waiting" to poll the variables while waiting for a transition to become enabled.

The Process task algorithm described below is similar to the Worker task functions (Chapter 4):

Loop forever

- 1) Detect a ready transition for the process.
- 2) Wait at the MC_TASK until allowed to execute.
- 3) Execute the transition.
- 4) Report back to the MC_TASK when finished.

End loop;

Channels between process tasks are implemented by direct rendezvous. If PROCESS_A wishes to initiate a primitive in CH_1, then PROCESS_A calls the RX_B rendezvous on PROCESS_B.

The channel implementation is a major flaw in the design due to potential deadlock. A deadlock situation exists if two processes attempt to initiate primitives to send messages to each other simultaneously. In this case, both tasks are blocked indefinitely waiting at each others rendezvous. The suggested

solution is to implement the initiation of primitives by a non-blocking mechanism.

5.3.2) Task Per Process: Version 2

The implementation of a non-blocking primitive activation may be accomplished by introducing active components in the channels. A simple solution is to use transport tasks as shown in Figure 5-4. The diagram only shows a channel implementation between two process tasks since the remainder of the model is not changed.

Consider the channel interactions at PROCESS_A. The TO_A transport task sends messages to PROCESS_A from PROCESS_B. The TO_B transport task sends messages from PROCESS_A to PROCESS_B. The TX_B and RX_B rendezvous are selectively accepted and are used to interface the transport tasks to PROCESS_A.

Sending a message to PROCESS_B involves opening the condition guard on TX_B. TO_B then makes a rendezvous at TX_B, takes the message and delivers it to PROCESS_B. The condition guard is closed on TX_B after TO_B takes the message. When TO_B returns to the rendezvous, the task will be blocked until another message is ready and the condition guard is re-opened. A queue internal to PROCESS_A must be maintained at TX_B to queue messages which may become ready while TO_B is busy delivering an earlier message. The condition guard on TX_B is closed while the queue is empty.

Receiving a message from PROCESS_B is done by accepting the

RX_B rendezvous. The TO_A transport task will rendezvous when a message is ready.

The use of the RX_B rendezvous to receive messages at a channel interface introduces polling problems. The Estelle channel model allows the message at the head of the channel queue to be polled but not removed from the channel. In contrast, once PROCESS_A accepts the RX_B rendezvous the message from the TO_A transport task must be dealt with.

The initial polling problem can be overcome by a more complicated programming of the RX_B rendezvous. The "intelligent" rendezvous would accept and examine the parameters from the transport task and then return a parameter indicating whether the transport task should get the next message from the channel or attempt the rendezvous again with the same message. The intelligent rendezvous allows the process task to poll the message contents and decide whether to consume the message immediately or accept the same message later.

Additional polling complexity arises when the process task must accept a unique rendezvous for each connected channel. All connected channels must be polled to determine if other transitions are enabled by other channel interactions. All enabled transitions must be identified to ensure fair resolution of non-determinant choices.

The polling complexity can be reduced by internally buffering

the message from the head of the channel queue. Only the head of the queue needs buffering since Estelle channels are currently implemented with a FIFO scheme. The process task can then use messages as needed and only accept a channel rendezvous to receive a new message if the internal buffer is empty. The solution also negates the need for the more complicated "intelligent" rendezvous.

The resulting channel implementation is not attractive for use as a protocol test vehicle. The contents of a channel are spread over the internal queue in the sending task, the channel transport task and a buffer internal to the receiving process task. Extraction of the contents of a channel during testing and verification would require accessing multiple objects.

The access to channel information would be more convenient if channels were implemented using centralized data structures.

5.3.3) Task-Per-Process: Version 3

The information distribution associated with using active channels could be eliminated if channel messages are queued in passive queue packages. The execution implication of using passive channels is the shifting of some of the responsibility for identifying enabled transitions from the Process tasks to the MC_TASK. The MC_TASK must now poll the channels connected to a process to determine if a message is pending.

The resulting model is now identical to the Dispatcher/Worker model with the exception of task assignment. The MC_TASK and the Dispatcher task are identical except for the mechanism of controlling execution of modules. The Dispatcher manages a single rendezvous for all Workers while the MC_TASK must manage a unique rendezvous for each Process. The Process tasks provide the same functionality as the Workers. The Process tasks are statically assigned to a module while the Workers are dynamically assigned. The Channel packages are virtually identical. It must be noted that the actual number of rendezvous executed is the same for both models.

The decision to have statically assigned Process tasks for each module or dynamically assigned Workers will not influence the performance of the execution model as described below.

5.3.4) Justification for Tasks

The execution testbed architecture is the determining factor in justifying the existence of a multitasking (as opposed to a sequential) execution model. The software engineering advantage of using tasks to specify logical concurrency is not a factor since modules accomplish this at the higher level specification stage.

If a single processor with an event driven tasking scheme is used, then real concurrency never exists and the execution of transitions could be performed sequentially by the Dispatcher or master control "tasks" which would really be acting as cyclic

schedulers. The use of tasks and a multitasking kernel will only slow the execution speed of the models on a uniprocessor testbed architecture. Tasks will provide an advantage to the execution model if executing in a multiprocessing environment.

5.4) Mapping of Estelle Specifications into Software Designs

An original goal of the thesis was to align protocol specifications with implementation software designs. Although the Dispatcher/Worker execution model provides one possible software design, the requirements of a real target system have not been met. Buffer management, timeout handling and real-time interactions are examples of requirements which distinguish an implementation independent specification from an implementation dependent software design.

This section identifies issues in the alignment of specifications with implementation designs. The issues are presented as unresolved questions which are intended to identify areas for future work. The discussion following each question ties in the material of earlier sections and presents the author's opinions where appropriate.

- (a) Is it desirable to use the protocol specification as a framework for the software design?

From a software engineering perspective it is desirable to use the original specification as the framework for the designs. An ideal alignment would allow the results of tests performed on the specification to be applied to the software designs. Failing this, at least a similar type of verification procedure should be possible for both the protocol and the implementation.

(b) Is it possible to use the protocol specification as the framework for ensuing designs?

If the protocol specification is to be used as an initial framework for the software design then a straightforward mapping must exist between objects in both the specification and implementation languages. Furthermore, the resulting framework must provide a convenient integration mechanism for introducing implementation details.

The feasibility of using the specification as a framework for the software design is a central issue to all of the questions in this discussion.

(c) Is the Dispatcher/Worker model a good software design?

The Dispatcher/Worker model uses the Estelle specification as a framework for software design. The modeling of Estelle execution behaviour in Ada has resulted in a poor software

modularization.

Managing the HER requires the Worker tasks to rendezvous with the Dispatcher before and after each transition is executed. A preferred modularization would have tasks which execute state machines without having to synchronize with the Dispatcher. The tasks would be active only when a state machine is ready to execute and the blocking and unblocking of tasks would be caused directly by protocol related events. The responsibility of synchronization and scheduling should not be forced into the protocol related application tasks. To implement HER behaviour, the functionality provided by the Dispatcher should be inherent in the execution of the tasks and controlled by the underlying operating system.

The Dispatcher/Worker model suffers (as a software design) from difficulties in aligning Estelle execution behaviour with Ada execution behaviour. The relationships between Estelle and Ada are discussed further in (f), (g) and (h).

(d) When should implementation details be introduced to convert a protocol specification into a software design?

The use of Caede for both protocol specification followed by software design closely parallels the "protocol engineering life cycle" presented by Wheeler et al [8]. The life cycle orders the milestones in protocol engineering as: the implementation

independent protocol specification, the implementation independent design and finally the implementation dependent design.

Similar milestones have been met using Caede. The modularization problem discussed in (c) prevents the execution model from being integrated into a feasible methodology, however the concepts behind the life cycle milestones have been illustrated. If an alternate execution model can be developed which overcomes the modularization problem then the life cycle would become a feasible approach using Caede.

To illustrate the milestone concepts in Caede, the initial Estelle diagrams provide the implementation independent protocol specification. The mapping of the Estelle specification to the Dispatcher/Worker model provides an implementation independent software design. The complete alignment would require the Dispatcher/Worker model to be extended to become the implementation dependent software design.

(e) How should implementation details be introduced to convert an implementation independent design into a target implementation?

For consistency in the development of designs, the extension of the implementation independent design should be carried out using objects similar to those used in the original specification and the implementation independent design. This approach was

successfully taken to expand an RMS state machine specification into an implementation design [23]. Both the specification and implementation were described using the same graphical notation. The extension of the original specification required the introduction of new implementation dependent states in the original transitions. The work in [23] did not include a mapping of the state machine design to a target architecture (hardware, operating system and language).

The concept of extending the software design using object consistent with the specification may be illustrated with Caede. The extension of the Dispatcher/Worker design into a target software design involves the introduction of additional information to support implementation dependent issues like buffer management, timeout handling and real-time interactions. To satisfy the proposed approach, the details of the implementation dependent issues must be specified using Caede diagrams.

A typical example taken from the RMS Service design involves the handling of a static buffering scheme. A user message may not be sent to the link layer by the Service SEND process (Figure 3-7) until a buffer is available. To implement the buffer management scheme, a new BUFFER_MANAGER module must be connected to the SEND module and the resulting SEND process state machine is shown in Figure 5-5. Additional state and event couplings have been introduced in the tx_msg transition to request and wait for a buffer before proceeding to send the message.

The example has shown that implementation details may be added in a consistent fashion using specification oriented objects. The construction of a Dispatcher/Worker model to execute the modified RMS design would treat the new BUFFER_MANAGER module and its connection to the SEND module in a fashion consistent with the protocol related modules. The modified SEND process state machine package would be constructed in an identical manner as all other state machine packages. The introduction of the implementation dependent buffer management scheme has not modified the construction technique for the execution model.

(f) Is there a difference between specification and implementation languages?

In terms of specification/design constructs, both Estelle and Ada contain similar abstract objects to describe structure and interconnection relationships. Modules and packages allow abstraction and structural partitioning. Processes and tasks provide concurrency. Channels and rendezvous allow communication between concurrent objects. Pascal and Ada support the definition of abstract data types and program actions. The fundamental difference is in the goals of the languages. The different goals lead to complexity in the alignment of Estelle specifications with Ada designs.

The goal of Estelle is to provide unambiguous specifications of protocols. Implementation of a target design is not a goal of

Estelle. The Estelle language specification [3] clearly states that implementations based on an Estelle description are not even required to conform to the internal modularization presented in the specification. The only requirement is that the outermost interface behaviour be the same. Software designers are left with a free hand to create an implementation architecture. Furthermore, Estelle does not provide for low level details like real-time event handling.

In contrast, Ada supports both specification and implementation. The use of Ada for specification provides an automatic mapping to an implementation since identical objects are used at both stages. Furthermore, Ada contains all implementation details needed for a target design.

When compared to Estelle, an apparent shortcoming of Ada is the absence of specific protocol oriented objects like state machines and channels. This is not a major limitation. The RMS example in section 4.3 has shown that state machines can be readily implemented in Ada and an architecture for channels has also been proposed.

(g) Is it necessary to have separate specification and implementation languages?

The mapping of specification objects to implementation

objects is readily accomplished if the specification is created using an implementation language. This implies that the specification would be directly executable. There would be no mapping problems since the specification and implementation design objects are the same. As previously stated in (f), Ada uses this approach.

Objections to this approach are that the specification should be implementation independent and the use of an implementation language to specify a protocol may force the direction of the implementation. Furthermore, the details of an implementation language are not important for the purposes of protocol specification. Large languages like Ada would present an unnecessary learning curve to a specifier who must then extract a useful vocabulary.

Caede's original approach used Ada as both a design and implementation language. The problem of language complexity was dealt with by selecting a subset of the complete Ada language for use at the specification stage.

The alternate approach taken in this thesis is to specify the protocol in a specification language and then map the specification to an implementation language. The approach is desirable since it allows all implementation concerns to be deferred until the appropriate phase and provides a very portable specification.

The objection to the approach taken is that the mapping from the specification to a feasible design may be complex. The mapping of Estelle to the Dispatcher/Worker model illustrates that the desire to avoid influencing the design during specification may result in a specification which is not useful as an implementation design template.

(h) Would an Estelle specification be easily mapped to a software design if some modifications were made to the execution behaviour?

The mapping of Estelle to Ada was severely complicated by the HER. Managing the HER caused further complexity in determining when a process was ready to make a transition.

These problems can be solved in two ways. One way is to modify Estelle's execution behaviour to more closely parallel an existing implementation language. The other way is to create a new execution language as an extension to Estelle. In light of the massive work required in the construction of a comprehensive language like Ada, the former alternative is more attractive.

Elimination of the HER from Estelle would require the introduction of a new mechanism to restrict child process transitions while a parent process is executing dynamic operations

on objects. The definition of an Estelle rendezvous may provide some of the required synchronization. The need for the HER is completely eliminated if dynamic operations on objects are not allowed.

Further modifications to Estelle might include implementation oriented enhancements to allow processes to be assigned priorities and restrictions on the use of exported variables in spontaneous transition enabling conditions. The assigning of priorities to processes would allow the specifier some control in the resolution of non-determinant choices between processes which simultaneously have transitions enabled. Most multitasking languages/operating systems allow process priorities to be assigned.

(i) Is the degree of concurrency exposed by the specification an optimum mapping for the implementation language?

This issue has not been explored and is left for future work.

(j) Can an alignment of the specification and software objects be performed automatically or are additional design decisions required?

Ideally, the conversion from a specification to an implementation would be automatically done. The implications of

the previous issues will have a definite impact on the success of automatic design generation.

5.5) Enhancements to Caede's Automated Support

This section has been included to record some of the author's ideas on improving Caede. The points raised in this section do not contribute to the ability to represent Estelle graphically, but the section does provide some areas for future development in Caede.

The use of Caede during the alternating bit and RMS case studies revealed some areas where Caede's automated support might be enhanced. The two proposals in this section deal with Caede's management of icons, diagrams and views. The first enhancement requires the handling of side effects when modifying diagrams. The editing of one diagram may have side effects on other diagrams. The proposed enhancement is to identify side effects automatically and to indicate where further corrections are needed. The second enhancement involves the handling of views. A more flexible mechanism is suggested.

The incorporation of textual input also requires enhancements to the Caede human interface. As mentioned in the previous section, support for the input of text is left for future work.

5.5.1) Side Effects of Diagram Modifications

A major step in the development of Caede would be the enhancement of the management of internal data structures to handle the side effects of modifying diagrams. For the purposes of constructing a diagram, Caede maintains an internal data structure which records the presence and inter-connection of icons on a single diagram. Additional relationships exist between diagrams when the contents of one diagram is represented in some form on another diagram. Side effects may occur when one of the diagrams is modified such that the other diagram is also influenced. Caede currently manages the links between diagrams to a very limited degree and the monitoring and correction of side effects must be handled by the specifier. Enhancements to provide automated management of side effects would free the specifier from this burden.

To illustrate the influence of side effects, consider the situation where the role primitives for a channel type require modification. For discussion purposes, assume that the R_request primitive was removed from the user interface port on the U_ACCESS_POINT channel template shown in Figure 2-3. Although the template is the only diagram modified, all instances of the channel and associated interface ports are influenced. For the example given, the channel instance is the U channel shown in Figure 2-4 and one of the influenced ports is the ALT_BIT_1.U_provider port. The R-request port plug is no longer present and therefore the side effect is felt in the need to modify the state machine and couplings of Figure 2-8.

The proposed enhancement is to build internal data structures into Caede which record relationships between diagrams. The data structures should allow Caede to monitor potential side effects automatically. Furthermore, Caede should prompt the specifier as to all points where side effects require further diagram modifications.

For the example given above, an internal data structure would record all diagrams showing instances of the U_ACCESS_POINT channel and interface ports. When the template was modified to remove the R_request primitive, all instances of the interface ports would be checked automatically by Caede. If there were any couplings between state machines and instances of the plug then Caede would prompt the specifier with a message indicating that additional diagrams require attention. When the diagrams containing the side effects are displayed a prompt should be issued and the icons influenced should be highlighted to draw the specifier's attention.

5.5.2) Views

An essential enhancement is required in the support of views. Sub-views are intended to capture information related to a specific aspect of the master view. Currently the master view of an object is the only view where the specifier is allowed to make modifications. All sub-views require the filtering of the master view. Any additions to the master view automatically appear on

all sub-views until filtered.

The current support is undesirable since sub-views must be continually filtered to remove unrelated information added to the master after the creation of the sub-view. The master view may also become excessively cluttered and difficult to work with.

The proposed enhancement is to reverse the relationship between the master view and sub-views. Individual sub-views would remain static unless information is specifically added or filtered in. Modifications to sub-views would automatically be reflected in the master. This is easily accomplished if the master view is automatically maintained as the overlay of all sub-views. Editing of the master view should still be allowed to provide a base from which sub-views can be initially constructed. This enhancement would eliminate both the view filtering and clutter problems associated with editing only the master view.

Figure 5-1: Internal Structure of LINK Module

Figure 5-2: Direct Coupling of SEND and S_QUEUE Modules

(a)

(b)

Figure 5-3: Example of the Task-Per-Process Execution Model

Figure 5-4: Active Channels for the Task-Per-Process Model

MOD_SEND

Figure 5-5: Modified SEND State Machine

6. Conclusions

The graphical notation currently integrated into Caede has been successfully used to specify the alternating bit (section 2.1) and RMS (Chapter 3) protocols. The application of the notation has illustrated the concept of representing Estelle graphically. Some specification information including abstract data types and program actions must be added as text. The recommended enhancements to the graphical notation presented in section 5.1 would be required to give a more complete representation of Estelle.

The Dispatcher/Worker model presented in section 4.2 provides an execution model for Estelle specifications. The guidelines presented in section 4.3 support automatic generation of the components in the model.

A feasible alignment of Estelle specifications with Ada implementation designs has not been accomplished due to difficulties in mapping the Estelle execution model behaviour into Ada. The issues yet to be resolved to provide a more general and complete alignment are identified and discussed in section 5.4.

In the opinion of the author, the original approach taken by Caede to use Ada as both a specification and implementation language is more promising than the alignment of Estelle with Ada. Further work is suggested in mapping concurrent state machines to Ada execution models. This would follow the lead of [23] and

incorporate the experience gained through the Estelle/Ada alignment.

Caede provides a framework for the alignment of the protocol specification process and the software engineering. This has been shown through the similarities in methodologies, the graphical representation of high level abstract objects and the successful use of Caede's notations to create both protocol specifications and designs.

In addition to the issues identified in section 5.4, the validation of protocols within the Caede framework is a major component for future research. The Caede tool set for protocol development will not be complete until a validation tool has been integrated with the graphical approach.

REFERENCES

- [1] R.J.A. Buhr, G.M. Karam, C.J. Hayes, C.M. Woodside, "Software CAD: A Revolutionary Approach", Technical Report SCE-86-13, Carleton University, Ottawa, Ontario, Canada, June 1986.
- [2] R.J.A. Buhr, C.J. Hayes, G.M. Karam, E. Neron, R. Casselman, "Caede 1.5 User's Guide", Technical Report SCE-86-11, Carleton University, Ottawa, Ontario, Canada, May 1986.
- [3] ISO, "Estelle - A Formal Description Technique Based on an Extended State Transition Model", ISO Draft Proposal ISO/DP 9074 97/21 N422, 1985.
- [4] U.S. Department of Defense, "Reference Manual for the Ada Programming Language", (MIL-STD-1815a), 1983.
- [5] CCITT, SDL Newsletter, No. 7, August 1984, containing the Z.100 to Z.104 series of specifications on SDL. Edited by F. Belina, Telelogic AB, Baltzarsgatan 22, S-211 36 MALMO, Sweden. Distributed by the Swedish Telecommunication Administration.
- [6] J.L. Peterson, "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, September 1977, pp 223-251.
- [7] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, Volume 92, Springer Publishing Company, Berlin, 1980.
- [8] G.R. Wheeler, T.J. Batten, J. Billington, M.C. Wilbur-Ham, "A Methodology for Protocol Engineering", Proceedings of the 9th International Conference on Computer Communication, October 1986.
- [9] A. Giacalone, J.D. Kowacs, "IDCCS: An Ideographic Syntax for CCS", Technical Report No. CS-83-05, Brown University, February 1983.
- [10] C.J. Fidge, G.J. Cain, L.N. Jackson, R.S.V. Pascoe, "An Overview of the MELBA Automatic Code Generation Project", A.T.R. Vol. 18, No. 1, 1984 (Australia).
- [11] R.E. Fairley, "Software Engineering Concepts", McGraw-Hill Book Company, 1985.
- [12] R.J.A. Buhr, "Systems Design With Ada", Prentice-Hall, Englewood Cliffs, N.J., 1984.

- [13] R.J.A. Buhr, C.M. Woodside, G.M. Karam, K. Van der Loo, D.G. Lewis, "Experiments with Prolog Design Descriptions and Tools in Caede: an Iconic Design Environment for Multitasking, Embedded Systems", Proceedings of the Third International Conference on Software Engineering, August 1984.
- [14] G.V. Bochmann, C.A. Sunshine, "Formal Methods in Communication Protocol Design", IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp 624-631
- [15] G.V. Bochmann, "A General Transition Model for Protocols and Communication Services", IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp 643-650.
- [16] Ansart, Amer, Chari, Lenotre, Mariani, Mattera, "Software Tools for Estelle", Draft Proceedings of the Sixth International Workshop on Protocol Specification, Testing and Verification, June 1986, pp 2-52 to 2-60.
- [17] G. Bruno, G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.
- [18] ISO, "Definition of the Temporal Ordering Specification Language LOTOS", Draft Proposal ISO/DP 8807 97/21 N423, April 1985.
- [19] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, Volume 92, Springer Publishing Company, Berlin, 1980.
- [20] J.P. Briand, M.C. Fehri, L Logrippo, A. Obaid, "Executing LOTOS Specifications", Draft Proceedings of the Sixth International Workshop on Protocol Specification, Testing and Verification, June 1986, pp 3-1 to 3-19.
- [21] J. de Meer, "Derivation and Validation of Test Scenarios Based on the Formal Specification Language Lotos", Draft Proceedings of the Sixth International Workshop on Protocol Specification, Testing and Verification, June 1986, pp 6-13 to 6-34.
- [22] ISO, "Estelle - A Formal Description Technique Based on an Extended State Transition Model", Report ISO/TC97/SC16/WG1 Nxxx, March 1984.
- [23] R.J.A. Buhr, T. W. Pearce, "Towards Designing Protocol Software with Caede", Technical Report SCE-85-13, Carleton University, Ottawa, Ontario, Canada, June 1985.
- [24] R.J.A. Buhr, T.W. Pearce, "RMS Specification", Technical Report SCE-86-16, Carleton University, Ottawa, Ontario, Canada, August 1986.

- [25] C.M. Woodside, R.J.A. Buhr, T.W. Pearce, "The Caede/Estelle Graphical Tool for Formal Definition of Protocols", Technical Report SCE-86-2, Carleton University, Ottawa, Ontario, Canada, March 1986.
- [26] C.M. Woodside, R.J.A. Buhr, T.W. Pearce, "The Caede/Estelle Graphical Tool for Formal Definition of Protocols", Proceedings of the Sixth International Workshop on Protocol Specification, Testing and Verification, June 1986.
- [27] T.W. Pearce, R.J.A. Buhr, "Caede/Estelle Case Study: RMS", Technical Report SCE-86-17, Carleton University, Ottawa, Ontario, Canada, August 1986.
- [28] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", Invited Paper, IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp 425-432.
- [29] A.A.S. Danthine, "Protocol Representation with Finite-State Models", IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp 632-643.

APPENDIX A

Estelle Specification of the Alternating Bit Protocol

This appendix contains the Estelle code for the specification of the Alternating Bit Protocol. The specification was originally presented in [22] and has been reproduced here for convenient reference. The specification was used during early experiments with mapping the state machine notation developed in [23] into Estelle. The Caede diagrams presented in Chapter 2 of this thesis are based on the specification.

Specification Example: the alternating bit protocol

Specification Example:

(* This is the top level module body *)

type

Cep_type =
Data_type =
Seq_type = (* for alternating bit, use 0 1 *)
Id_type = (DATA, ACK);
Ndata_type = ..

record

Id: Id_type;
Conn: Cep_type;
Data: Data_type;
Seq: Seq_type;

end;

(* Channel definitions for communication between the processes *)

channel U_access_point(User, Provider):

by User:

SEND_request (Udata: Data_type);
RECEIVE_request;

by Provider:

RECEIVE_response(Udata: Data_type);

channel N_access_point(User, Provider):

by User:

DATA_request(Ndata: Ndata_type);

by Provider:

DATA_response(Ndata: Ndata_type);

(• Module header definitions •)

module User_type process
 (U: U_access_point(User) common queue;
 param Name: Cep_type);

module Alternating_bit_type process
 (U: U_access_point(Provider) common queue;
 N: N_access_point(User) common queue;
 param Name: Cep_type);

module Network_type process
 (N: array [Cep_type] of N_access_point(Provider) common queue)

(• Body definitions •)

body Network_body for Network_type: external;

body User_body for User_type: external;

(* the body for alternating bit is defined below *)

body Alternating_bit_body for Alternating_bit_type:

const

Retran_time = (* retransmission time *)

Empty = (* empty buffer *)

type

Msg_type

record

msgconn: Cep_type;

Msgdata: Data_type;

Msgseq: Seq_type;

end;

Buffer_type

Int_type = (* would usually be integer *)

(* Channel definition *)

channel S_access_point(User, Provider):

by User:

TIMER_request;

by Provider:

TIMER_response;

(* Activity module definitions within Alternation_bit_body *)

module Data_transfer_type activity

(U: U_access_point(Provider) common queue;

N: N_access_point(User) common queue;

S: S_access_point(User) individual queue;

param number: Cep_type);

module Timer_type activity

(S: S_access_point(Provider) individual queue;

param Time: integer);

(* The body for data_transfer_activity is defined below *)

body Data_transfer_body for Data_transfer_type:

var

Send_seq:	Seq_type;
Recv_seq:	Seq_type;
Send_buffer:	Buffer_type;
Recv_buffer:	Buffer_type;
P.Q:	Msg_type;
B:	Ndata_type;

state: (ACK_WAIT, ESTAB);

stateset

EITHER = [ACK_WAIT, ESTAB];

pure function Ack_ok(Nd: Ndata_type): boolean;

begin

Ack_ok := (Nd.Id = ACK) and (Nd.Seq = Send_seq);

end;

procedure Format_data(Msg: Msg_type; var B: Ndata_type);

begin

B.Id := DATA;

B.Conn := Number;

(* connection reference given in the instantiation *)

B.Data := Msg.Msgdata;

B.Seq := Msg.Msgseq;

end;

procedure Format_ack (Msg: Msg_type; var B: Ndata_type);

begin

B.Id := ACK;

B.Conn := Msg.Msgconn;

B.Data := Msg.Msgdata;

B.Seq := Msg.Msgseq;

end;

procedure Store(var Buf: Buffer_type; Msg: Msg_type);
primitive;

procedure Remove(var Buf: Buffer_type; Msg: Msg_type);
primitive;

function Retrieve(Buf: Buffer_type) Msg_type;
primitive;

function buffer_empty(Buf: Buffer_type) : boolean;
primitive;

procedure Inc_send_seq;
begin
 Send_seq := (Send_seq + 1) mod 2
end;

procedure Inc_recv_seq;
begin
 Recv_seq := (Recv_seq + 1) mod 2
end;

initialize ('For the data-transfer-activity ')

to ESTAB (' initialize state and variables ')

begin
 Send_seq := 0;
 Recv_seq := 0;
 Send_buffer := Empty;
 Recv_buffer := Empty;

end;

(* Transitions for the data-transfer-activity *)

trans

from ESTAB

to ACK_WAIT

when U SEND_request

begin

(* transition 1 *)

P.Msgdata := Udata;

P.Msgseq := Send_seq;

Store(Send_buffer, P);

Format_data(P, B);

output N.DATA_request(B);

output S.TIMER_request

end;

from EITHER

to same

when U RECEIVE_request

provided not buffer_empty(Recv_buffer)

begin

(* transition 2 *)

Q := Retrieve(Recv_buffer);

output U.RECEIVE_response(Q.Msgdata);

Remove(Recv_buffer, Q)

end;

from ACK_WAIT

to ACK_WAIT

when S TIMER_response

begin

(* transition 3 *)

P := Retrieve(Send_buffer);

Format_data(P, B);

output N.DATA_request(B);

output S.TIMER_request

end;

from ESTAB

to ESTAB

when S TIMER_response

begin

(* transition 4 *)

(* Do nothing: the message that caused this
timer to be set has been acknowledged. *)

end;

from ACK_WAIT

to ESTAB

when N.DATA_response

provided Ack_ok(Ndata)

begin

(* transition 5 *)

```

P := Retrieve(Send_buffer);
Remove(Send_buffer, P);
Inc_send_seq

```

```

end;

```

```

from EITHER
to same

```

```

when N.DATA_response

```

```

provided Ndata.Id = DATA

```

```

begin

```

```

(transition G)

```

```

Q.Msgdata := Ndata.Data;

```

```

Q.Msgseq := Ndata.Seq;

```

```

Format_ack(Q.B);

```

```

output N.DATA_request(B);

```

```

if Ndata.Seq = Recv_seq then

```

```

begin

```

```

Store(Recv_buffer, Q);

```

```

Inc_recv_seq

```

```

end

```

```

end;

```

```

end; (This is the end of the data transfer activity)

```

(* The body for timer activity is defined here *)

body Timer_body for Timer_type:

var

Stop, Stop_bis: boolean;

initialize

begin

Stop := true;

Stop_bis := true;

end;

trans

when S.TIMER_request

begin

(* transition 1 *)

(* Cancel previous timer *)

Stop := true;

Stop_bis := false;

end;

trans

provided not Stop_bis

begin

(* transition 2 *)

Stop_bis := true;

Stop := false;

end;

provided not Stop

delay (Time, Time)

begin

(* transition 3 *)

Stop := true;

output S.TIMER_response

end;

end. (* End of the timer activity body *)

(* End of module-declaration-part of the Alternating_bit_body *)

var

Data_transfer: Data_transfer_type;
(* for the Data_transfer activity *)

Timer: Timer_type; (* for the timer activity *)

initialize (* initialization-part of Alternating_bit_body *)

begin

init Data_transfer with Data_transfer_body(Name);

init Timer with Timer_body(Retrans_time);

connect Data_transfer.S to Timer.S;

attach U to Data_transfer.U;

attach N to Data_transfer.N

end;

end; (* End of the Alternating_bit_body *)

(* Alternating_bit_body has no transition part *)

(* The module_declaration_part of the specification is completed *)

(* variable_declaration_part of the specification *)

var

User: array [Cep_type] of User_type;

Alternating_bit: array [Cep_type] of Alternating_bit_type;

Network: Network_type;

(* initialization part of the specification *)

initialize

begin

init Network with Network_body;

all Cep: Cep_type do

begin

(* module initializations *)

init User [Cep] with User_body(Cep);

init Alternating_bit[Cep] with

Alternating_bit_body (Cep);

(* connection part *)

connect User[Cep].U to Alternating_bit[Cep].U;

connect Alternating_bit[Cep].N to Network.N[Cep];

end;

end;

end. (* End of specification;

The specification has no transition part *)

END

08.08.87

FIN

Channels:

Channels are an abstract mechanism used by Estelle to allow modules to communicate. A channel is characterized by a set of communication primitives which are distributed between two complimentary roles. A channel may be used by two modules to communicate if one module is assigned one role and the other module is assigned the complimentary role. The channel behaves as if each role is supported by an infinite queue. The activation of a primitive at one role results in a message being placed in the queue at the complimentary role. A module connected to a role must process messages received in its role queue.

Module Structure:

Structural information is specified in Estelle through the nesting of modules and interconnection of modules via channels. Estelle allows the hierarchical sub-structuring of modules in the creation of a protocol specification. The highest level module is called the "specification" and all other modules are nested to various levels.

Channels may be connected between the external interaction points on pairs of peer modules. Channels may also be bound between the internal interaction points of a parent and the external interaction points of a direct child module.

Internal interaction points are required only in architectures where a parent module state machine needs to communicate with nested sub-modules. Communication between modules in different branches of the structural hierarchy tree requires the use of the "attach" operation to bind a child module's external interaction point to a parent module's external interaction point. The child may then communicate directly through the channel connected to the parent.

State Machines:

Estelle represents temporal behaviour through coupled state machines. A module may contain (at most) one state machine at the parent level. Concurrent behaviour must be decomposed into concurrent state machines with a module defined for each state machine. The coupling of state machines is accomplished through channels between the modules containing the state machines.

A state machine is a set of transitions which interconnect a set of states. A state machine waits in a state and executes actions during the transitions between states. The paths created through the states by the transitions specify all of the allowable temporal sequences which may be executed by a module.

Each transition has an enabling condition and an action block. The enabling condition must be satisfied before the

transition action block may be executed. The action block is assumed to be atomic and for the purposes of the state machine, the block must be completely executed once the transition is started. Once a state is entered, a state machine waits in the state until an event occurs which satisfies an enabling condition for one of the transitions originating from the state. The transition action is then executed which may lead the state machine to a new state or return it to the same state.

Coupling occurs when an action in one state machine must be communicated to another state machine. The information received through communication influences the enabling of transitions and thereby influences the resulting temporal sequence. Channels exist between modules containing coupled state machines. A state machine passes information by activating a channel primitive during the execution of an action block. An appropriate message is placed in the complimentary role queue which must then be processed by the coupled state machine.

END

08.08.87

FIN