A GRAPHICAL REPRESENTATION FOR THE FORMAL DESCRIPTION TECHNIQUE ESTELLE

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Justin George Templemore-Finlayson

July 1998

Supervised by

Prof P.S. Kritzinger

Abstract

This dissertation concerns the specification and description of complex communicating systems using Formal Description Techniques. Specifically, we propose a standard graphical representation for the Formal Description Technique Estelle and present a prototype editor based on this representation. Together they integrate the new graphical representation with existing Estelle textual tools to create a powerful graphical design technique for Estelle.

The perennial popularity of graphical techniques, combined with recent advances in computer graphics hardware and software which enable their effective application in a computing environment, provide a double impetus for the development of a graphical representation for Estelle.

Most importantly, a graphical technique is more easily read and understood by humans, and can better describe the complex structure and inter-relationships of components of concurrent communicating systems.

Modern graphical technology also presents a number of opportunities, separate from the specification method, such as hyperlinking, multiple windows and hiding of detail, which enrich the graphical technique.

The prototype editor makes use of these opportunities to provide the protocol engineer with an advanced interface which actively supports the protocol design process to improve the quality of design.

The editor also implements translations between the graphical representation and the standard Estelle textual representation, on the one hand allowing the graphical interpretation to be applied to existing textual specifications, and on the other, the application of existing text-based processing tools to a graphical specification description.

Acknowledgements

I would like to extend my sincerest thanks and appreciation to the following who made this all possible:

- My parents who although they are not with me, always are.
- Professor Pieter Kritzinger for his needed guidance, encouragement and support in uncharted waters.
- Professor Stan Budkowski and Jean-Luc Raffy, Estelle enthusiasts who started it all.
- The staff of the Departement Logiciels-Reseaux at the Institut National des Télécommunications who made it not only possible, but enjoyable to be there.
- The three good witches, Eve, Mary and Rowena, for their organisational sanity in the Computer Science department.
- Bernie, for the lots of little things.
- My fellow post-graduate students in Computer Science, for being crazy enough to follow the same route; I did not go it alone.
- Karim Chenikhar, PhD student at Electricité de France, for his helpful information on ASA+.
- Members of the Estelle user community for their comments and suggestions over the months.
- The French Government and South African Foundation for Research and Development, for sponsoring this research.

Contents

List of Figures

Chapter 1

Introduction

Formal Description Techniques (FDT) are important tools for the design and implementation of concurrent communicating systems (CCS). A CCS is any large, distributed system in which many components execute concurrently and inter-communicate. Such a system tends to exhibit unpredictable behaviors which can result in failures which can lead to loss of life, irreplaceable information or operating income. Typical concurrent communicating systems include network protocols, manufacturing systems, medical information systems and business processes.

Formal Description Techniques such as Estelle are able to *formally* define the complex behavior of a concurrent communicating system. This formal definition eliminates ambiguity from the system design, a common source of error. An formal definition can also be used as a basis for analysis of the correctness and performance of a system, and the generation of test cases to validate an implementation.

Formal Description Techniques in the computing environment have until recently been confined to a textual nature due to limitations of graphical interfaces which could not display the large canvases needed to describe a graphical system properly. Graphical techniques have always been preferred to textual techniques however, and used since before the advent of computing environments. Diagrams are quicker to read and comprehend by humans, and therefore a graphical description is usually preferred to text. Importantly, graphical techniques are accessible to a wider range of users, and extend the usefulness of the Formal Description Technique to non-academic users.

This preference for graphical techniques has often been cited as the reason for the greater popularity of the Specification and Description Language (SDL) compared to Estelle. These two FDTs are extremely similar, apart from the fact that SDL is a well-developed graphical technique, and Estelle has "only" a textual syntax.

Based on this motivation, we propose a formal graphical technique for Estelle.

We attempt to develop a *standard* representation based on convention. Standardization is essential to enable the development of tools which support the use of the graphical technique. The SDL technique is supported by at least two commercial toolkits, which are important factors in the widespread application of this technique.

Although no formal graphical Estelle techniques have been proposed, a limited number of tools have been developed which informally introduce useful graphical notations for Estelle. We incorporate these to make the technique familiar to established Estelle users. To ensure this representation remains a standard, it will be submitted for standardization to the International Standards Organisation (ISO), which maintains Estelle.

The second part of this dissertation presents a first prototype editor based on this graphical representation for Estelle.

The prototype editor takes advantage of the opportunities presented by recent advances in graphical computer interfaces to enrich the facilities of the graphical technique. Functionality such as multiple windows, hyperlinks between windows, level of detail selection and syntax-directed editing assist the user during the design process. The graphical editor compliments the graphical technique, and one of the features of the prototype editor is the ability to abstract details of the textual syntax where the graphical representation itself cannot.

A concurrent communicating system defined by a Formal Description Technique is also too complex to analyze manually, and therefore it essential to its effective use that it can be processed further automatically. The prototype editor once again plays an important role in this process, by translating a graphical Estelle description into an equivalent textual representation, which can be further processed by existing Estelle tools.

1.1 Dissertation roadmap

The remainder of this dissertation is organised as follows:

Chapter 2 presents background information on Formal Description Techniques. It focuses in particular on the similar techniques Estelle and SDL, and introduces the SDL graphical representation.

Chapter 3 is a tutorial introduction to the Estelle model. It is intended to give non-users of Estelle sufficient background on this technique to be able to understand and start using the graphical representation.

Chapter 4 surveys two existing graphical Estelle tools and analyses their graphical syntaxes.

Chapter 5 defines the formal graphical syntax and semantics of the Estelle graphical representation, and forms the main body of this dissertation. An example protocol specification - the Trivial File Transfer Protocol (TFTP) - is used to illustrate the representation.

Chapter 6 describes the features and functionality of a prototype editor which interprets the Estelle graphical representation. This includes a number of screenshots.

Chapter 7 draws conclusions on how the Estelle/GR and the editor extend the usefulness of the Estelle FDT.

Appendix A presents an implementation of the Trivial File Transfer Protocol as an example of the use of the Estelle/GR. The protocol was specified using the prototype editor, and the graphical diagrams and ISO Estelle code generated by the editor are reproduced.

Appendix B is for reference purposes and lists the collected Estelle/GR syntax from Chapter 5.

Chapter 2

Formal Description Techniques (FDT)

2.1 Introduction

Formal Description Techniques (FDT) are important tools for the design and implementation of concurrent communicating systems (CCS). A CCS is characterized by a high degree of complexity as it is a system with many components running in parallel and intercommunicating. Traditional software engineering methods are unable to describe the concurrency of a CCS, or make any guarantees about the operational reliability or performance of the system as a whole. The Formal Description Techniques discussed in this chapter are specifically designed for the definition of concurrent communicating systems, and the protocols for their intercommunication. The formal nature of the techniques allows a level of confidence to be expressed about the reliability of a system.

Large concurrent communicating systems such as public data switching networks are also characterized by services from multiple vendors. An FDT provides an unambiguous service requirements definition, which enables different services or components to be designed to inter-operate smoothly.

Estelle [ISO97], the Specification and Description Language (SDL) [ITU93b], LOTOS [BB87] and Message Sequence Charts (MSC) [ITU93a] are all Formal Description Techniques widely used to describe concurrent communicating systems. The Petri-net [Pet62] is another which

operates at a lower level of abstraction.

In this chapter we concentrate mainly on the very similar techniques **Estelle** and **SDL**. These are two Formal Description Techniques for **defining** the functional behavior of concurrent communicating systems based on the **extended finite state machine** model. The SDL has a graphical notation whereas Estelle does not.

2.2 Characteristics of Formal Description Techniques

An FDT differs from *informal* modeling techniques such as the Unified Modeling Language (UML) and general programming languages such as Pascal and C, in that the syntax is subject to a strict set of semantics which eliminates ambiguity. A formal description is thus self-contained as it does not rely on external knowledge of the described system for accurate interpretation. A formal technique is also usually rich in constructs, allowing system descriptions to be more concise than if a general language were used.

An FDT may also be based on a formal mathematical model. This enables the use of analytical methods to demonstrate the correctness of a formal description. Estelle and SDL, for example, are based on the model of an Extended Finite State Machine (see Section 2.4), and LOTOS is based on the theory of process algebras. EFSM, process algebras and Petri-nets all have formal mathematical models which can demonstrate the correctness and efficiency of a system.

Standardization is the key to unlocking the utility of Formal Description Techniques. International standards define the unique semantics which make a formal description complete, consistent, precise and unambiguous. Standards are maintained by international bodies such as the ISO or ITU, and any amendments to a standard requires international agreement. This keeps the Formal Description Technique stable and consistent over time.

Standardization of an FDT also facilitates the independent development of tools to support the technique. Support tools are invaluable as they simplify and automate the design and implementation processes. Well-developed toolkits such as Xedt [INT97] for Estelle, and ObjectGEODE [Ver96] and SDT [Tel98] for SDL, support the entire software engineering process from specification design to implementation.

Such toolkits create much added value for a Formal Description Technique. The toolkit

facilitates design through integrated environments which provide editing and debugging facilities; and implementation by automatic code and test case generation.

[Som92] summarizes the benefits of using Formal Description Techniques as follows:

1. The development of a formal description forces understanding of the system requirements. This reduces the chance of introducing design errors into a system.
2. An FDT is based on a mathematical model which allows verification of system consistency and completeness. It may also be possible to prove that an implementation conforms to its specification.
3. Formal descriptions may be automatically processed. Software tools can be built to assist and automate system implementation and debugging.

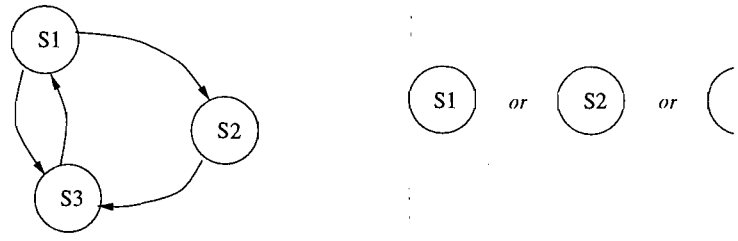
In addition, test cases can be generated from the formal specification which can be used to validate and verify implementations.

2.3 Definitive and descriptive techniques

Techniques in general can be used to either *define* or *describe* systems:

- **Definitive** techniques statically *define* the unique functional behaviour of a system.
- **Descriptive** techniques dynamically *describe* (as opposed to "define") a non-deterministic state or behavioral sequence of a system.

The difference is illustrated by the simple example of the static definition and dynamic descriptions of an automaton in Figure 2.1. The behaviour of the automaton is *defined* in 2.1(a) as a set of states $S1$, $S2$, or $S3$, with transitions indicating the ability to move between those states. At any discrete point in time however, that automaton can only be in one of $S1$, $S2$, or $S3$, as shown in 2.1(b), each of which constitutes a different description of the system. The example illustrates another difference between a definition and a description: A definition is unique, but may give rise to many descriptions.



(a) definition (b) descriptions

Figure 2.1: **Definition and description of a simple automaton.** The single definition in (a) gives rise to three possible states in (b).

The essential difference between the static definition and dynamic descriptions is that any description can always be derived from the static definition. Given a single dynamic description however, the static definition can not be deduced.

An example of the difference between definitive and descriptive Formal Description Techniques is the relationship between SDL and Message Sequence Charts (MSC). The SDL is a *definitive* technique which defines the functional behavior of a system An MSC is often used to describe a behavioral sequence of an executing SDL system and is therefore a *descriptive* technique. In addition, a single SDL system will be able to generate many MSC traces, while a single trace cannot be used to derive the original SDL specification.

Table 2.1 categorizes some modeling techniques as either descriptive or definitive.

Definitive	*Descriptive*
State-transition diagram	State
Estelle	LOTOS
SDL	Message Sequence Charts
Petri nets	Petri net markings

Table 2.1: **Some well-known definitive and descriptive modeling techniques.**

A *definitive Formal Description Technique* has a formal syntax and semantics, and can be used for formally defining concurrent communicating systems. This is not true of *descriptive Formal Description Techniques*, which can only describe behavior instances of a system, i.e.

some non-deterministic results of the execution of the static definition.

2.4 The Extended Finite State Machine (EFSM) model

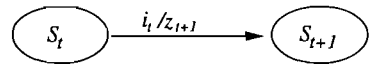
The extended finite state machine (EFSM) model is a formalism for describing discrete, event-driven systems. It is intuitive, yet with a well-developed mathematical foundation which makes it it ideal for describing information processing systems at a high level. Two of the most popular Formal Description Techniques in use today - Estelle and the Specification and Description Language - are both based on the EFSM model.

A finite state machine is a system of *states* and *transitions* between those states. A discrete set of states represent every possible state of the system, and transitions indicate the possibility to move from one state to another. Definition 1 is taken from [AH75] and defines a *Mealy* automata.

Definition 1 *A finite state machine is defined as a quintuple* $\{S, I, Z, \delta, \omega\}$*, where:*

$S = \{s_1, s_2, \ldots s_{\|S\|}\}$	*the discrete set of internal states of the automaton*
$I = \{i_1, i_2, \ldots i_{\|I\|}\}$	*the set of input messages, meaningful to the automaton*
$Z = \{z_1, z_2, \ldots z_{\|Z\|}\}$	*the set of output messages generated by the automaton*
$\delta : S \times I \rightarrow S$	*the 'next state relation' which defines the next state* S_{t+1} *on receipt of input* I_t *when in state* S_t
$\omega : S \times I \rightarrow Z$	*the function that defines the output* z_{t+1} *for a transition from state* s_t *on input* i_t*.*

This is represented graphically as a directed graph with a node for each state of S, and arcs representing transitions between states. An arc from state s_t to state s_{t+1} has a label of the form i_t/z_{t+1}, where i_t and z_{t+1} are determined by the functions δ and ω respectively.



Definition 2 *An extended finite state machine is a finite state machine that can use and manipulate data stored in variables local to that machine.*

An extended finite state machine may define data which is global to the machine, and extends the transition definition by the inclusion of programming language statements which manipulate this data.

2.5 Estelle

Estelle is a Formal Description Technique for defining distributed, concurrent communicating systems, specifically the protocols and services of the Open Systems Interconnection (OSI) Basic Reference Model [ISO94] defined by the International Standards Organisation (ISO). The technique was first standardized in 1989 after 10 years of study, and the latest standard document [ISO97] appeared in 1997.

An Estelle system is a hierarchy of inter-communicating components, each of which is defined by an extended finite state machine. Estelle is implementation-oriented. It defines the functional semantics of a system, as opposed to behavioral semantics as in a language like LOTOS It is implementation-oriented and it is possible to derive declarative instructions directly from the functional semantics. Estelle also uses object-oriented separation of interface and implementation definition which permits the user to determine the level of detail of the specification.

Estelle is currently used for the description of a wide range of telecommunications systems, from military to multimedia communication protocols. The main toolkit available for the language is Xedt [INT97], which is used in over 30 research and industrial centers around the world, including Alcatel and the United States military.

The Estelle model is discussed in detail in Chapter 3.

2.6 Specification and Description Language (SDL)

SDL is a widely recognised international standard for formally defining concurrent communicating systems, particularly communication services and protocols. SDL is maintained by the International Telecommunications Union (ITU), which has developed it from an informal notation for drawing process graph symbols, to the Formal Description Technique. It is based on the extended finite state machine model and incorporates abstract data types, object orientation and Message Sequence Charts that it is today.

SDL is used in more than 1000 companies worldwide. Users include the European Telecommunications Standards Insitute, AT&T, Ericsson, Alcatel, Renault, BT and many others. On of the reasons for the popularity of SDL is that it has a graphical syntax that provides the user with a clearer and more manageable description of the system being designed.

2.6.1 Introduction to the SDL graphical representation

The many common elements of Estelle and SDL, and the familiarity of the latter's representation in the formal description world, mean that it is not possible to ignore the SDL graphical representation (SDL/GR) when developing an equivalent for Estelle. The following is taken from [BH89], an introductory tutorial to SDL88, and introduces the SDL/GR.

2.6.1.1 Basic system structure

SDL has a hierarchical structure in which blocks are successively partitioned into smaller blocks in an effort to help cope with complexity. The SDL system-level contains one or more blocks, interconnected with each other and the environment by directed *channels* (see Figure 2.2). A channel is a means of conveying signals. These blocks can be infinitely partitioned into (sub)blocks and channels.

Repeated block partitioning results in a block tree structure with the system as the root block. Leaf blocks in this structure can be partitioned into *processes*, which define a behaviour. Within a leaf block, signals are conveyed on *signal routes*, which are analogous to *channels* (see Figure 2.3).

2.6.1.2 Process behaviour description

A process in SDL is an extended finite state machine, described using a flow diagram. In SDL there are five basic constructs for describing a process: *start, state, input, output* and *nextstate* which define the basic, *non*-extended finite state machine. Figure 2.4 shows the SDL/GR representation of these constructs.

Figure 2.5(b) shows the use of (syntactically correct) SDL/GR for describing the simple Mealy automaton presented in Figure 2.5(a). Two states *S1* and *S2* are involved. When in state *S1*, the machine is able to perform a transition to state *S2* on receipt of an input A,

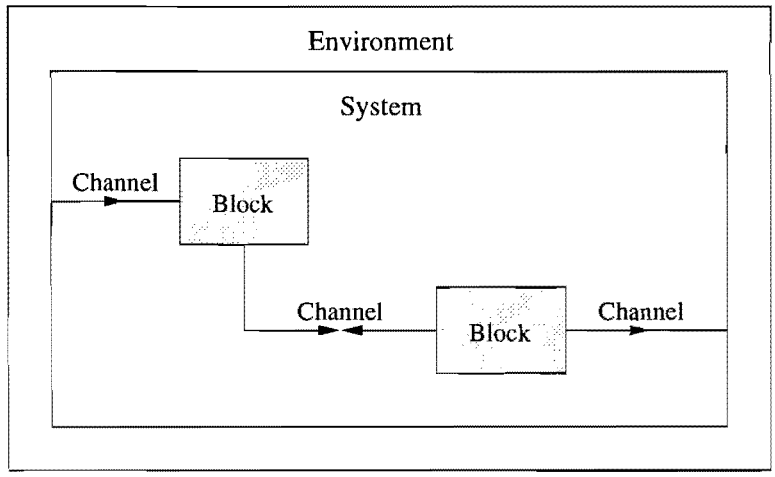


Figure 2.2: **SDL/GR: System block diagram.** Block partitioning helps to cope with complexity. Directed channels show in which direction messages may be sent.

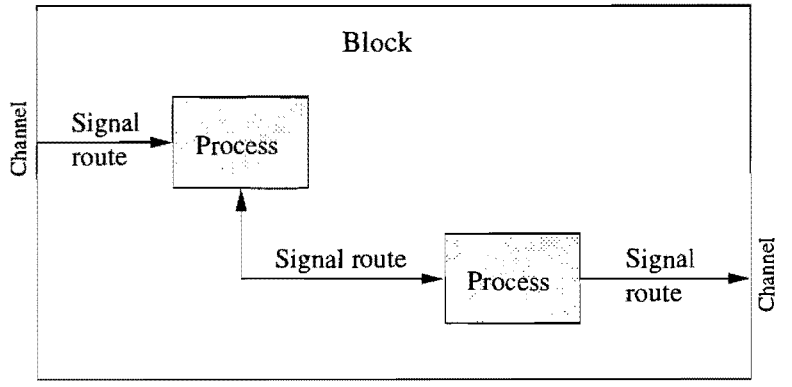


Figure 2.3: **SDL/GR: Structuring a block into processes.** Processes define behaviors which execute concurrently.

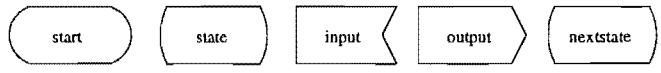
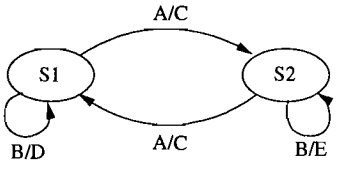
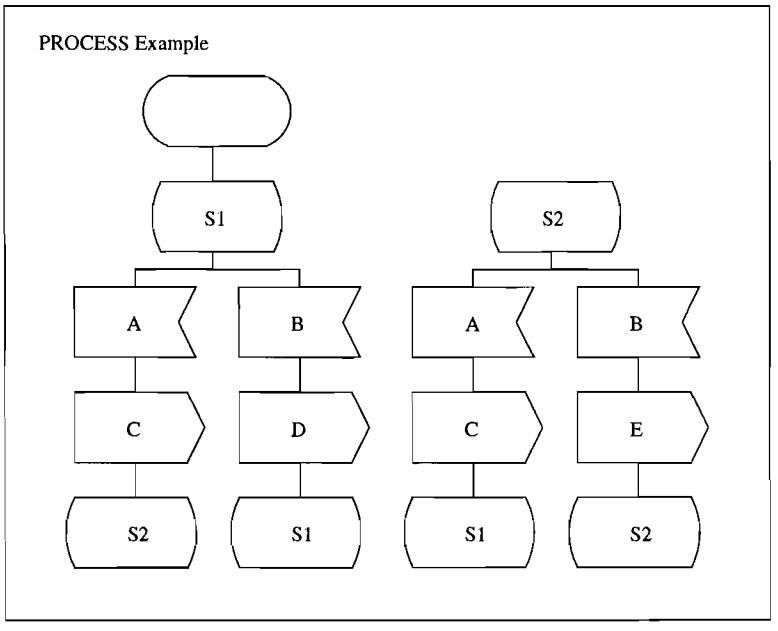


Figure 2.4: **SDL/GR: Basic constructs for the description of a process.**

during which it will produce an output C. The input B in state $S1$ results in a transition to the same state with output D, and so on.



(a) A simple finite state machine



(b) Description of the finite state machine of (a) by a process diagram

Figure 2.5: **SDL/GR: Process diagram of a simple automaton.**

The full formal definition of the SDL/GR can be found in the ITU Recommendation Z.100 [ITU93b]

2.7 Graphical techniques

Two Formal Description Techniques have been introduced in this chapter: SDL and Estelle. These techniques have much in common, but the greatest difference between them is the existence of a formal graphical technique for SDL, the SDL/GR. SDL also experiences far more widespread use than Estelle, and many in the industry cite the existence of the graphical technique as the reason for this greater popularity.

Graphical techniques have been popular since long before the development of digital computing systems. A simple diagram, plan or sketch is usually far more intuitive and easier to read than an equivalent textual description, which is often nothing more than a cumbersome description of the diagram. Limitations in early graphics hardware and software however inhibited the development of graphical techniques in the digital computing environment, and have only recently been overcome.

Consider for example, how a graphical (SDL) and textual (Estelle) technique differently describe the structure of a concurrent communicating system which defines a hierarchy of concurrently executing components.

The nested box notation of the SDL/GR (see Figures 2.2 and 2.3) exposes these relationships in a concise and intuitive manner. Hierarchical relationships are shown by graphically including child components within their parent definition, and as sibling components are placed adjacent to each other, may correctly be presumed to execute concurrently.

The concurrency information is lost in an Estelle textual description, in which sibling components are defined *sequentially* in the text. In addition, the syntax requires that child modules are completely defined within a parent module, which can result in huge module definitions and which obstructs the interpretation of the entire hierarchy.

The SDL graphical technique is not only more concise, but conveys substantial information as it separates the definition of the system structure from the behavior. In Estelle, the behavior is defined 'in-line' with the structure, resulting in unstructured text which conveys no obvious semantic meaning.

Limitations in early computer hardware and software encouraged the development of textual rather than graphical techniques however. Terminals were restricted to text only, and unable to support the large drawing spaces or "canvases" required to implement notations like the SDL/GR.

Recent advances in computer hardware and software graphics have now removed this impediment. Advanced user interfaces capable of providing multi-windowed environments and large, scrolling canvases make it possible to effectively implement computer-based graphical techniques. Furthermore, the emergence of languages dedicated to graphical user interface development such as Tcl/Tk, which was used to develop the prototype editor in Chapter 6, has simplified the task of implementing the advanced graphical features required.

Graphical user interfaces have also developed to a level of sophistication which, separate from the specification method, can in itself enrich the method through techniques such as using hyperlinks, zooming or hiding detail of the description.

These advances are enabling the widespread adoption of graphical techniques in the computing environment, and encouraging the graphical evolution of existing textual techniques such as Estelle.

Chapter 3

The Estelle model

3.1 Introduction

The exact syntax and semantics of Estelle are defined in ISO9074 Estelle standard [ISO97]. In this dissertation we refer to this *phrasal representation* as the **Estelle/PR**. Analogously the graphical representation presented is abbreviated as the **Estelle/GR**.

Semantically, an Estelle specification "describes a hierarchically-structured system of concurrently executing, non-deterministic sequential components interchanging messages through bi-directional links between their ports" [ISO97]. The system has a dynamic structure as both the hierarchy of components and the structure of communications links may change over time. The behavior of each component is defined as an extended finite state machine.

Syntactically, Estelle/PR is an *extended subset* of the general programming language Pascal [ISO83]. The technique extends the Pascal syntax with model-specific constructs, and also removes features of the general language which are inconsistent with the formal Estelle model.

The following discussion introduces the semantics of the Estelle model without going into the details of the Estelle/PR syntax. The objective is to familiarize the user with the Estelle semantics so that he or she can immediately start using the Estelle/GR without first having to learn the Estelle/PR.

This discussion of the Estelle model cannot help but duplicate some of the work of other excellent tutorials on Estelle, such as [BD87, Hog89], or indeed the ISO Estelle standard

[ISO97] itself. We add to the work of those authors in the following two ways:

- Introducing the model in a visual form.
- Emphasizing the dynamic structure of an Estelle system.

The motivation for the first point is to introduce - informally - some visualization of Estelle concepts before defining the formal graphical syntax and semantics in Chapter 5. It is also grounded in a firm belief in the expressive nature of diagrams.

Secondly, previous tutorials on Estelle have often overlooked the *dynamic structure* of an Estelle system, even though it is a unique and powerful feature of Estelle. In this section we describe in detail the dynamic nature of the Estelle module instance hierarchy and communication link structure. These are important aspects in the context of the Estelle/GR as they introduce substantial complexity into the notation.

3.2 The Estelle module concept

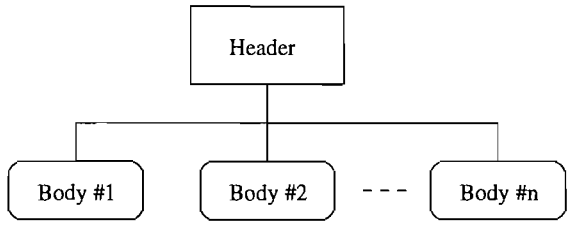


Figure 3.1: **A generic module definition.** A generic module definition consists of a single module header definition and one or more module body definitions. The header defines a consistent interface for instances created from this generic module, while each body defines an alternative internal behavior. The header is shown by a rectangle, an associated body by a rounded rectangle.

A component in an executing Estelle system is known as an *module instance*, and is defined statically by a **header definition** and a **body definition** associated with this header. More than one body definition may be associated with the same header, and this structure of one header and multiple associated bodies is known as a *generic module definition*. The concept is illustrated in Figure 3.1.

Several instances of the same generic module may be created and be simultaneously present during the execution of an Estelle specification. Each of these has the same external visibility, characterized by the module header definition, but potentially different internal behaviors, characterized by one of the associated module body definitions.

From an external viewpoint, a module instance is a 'black box', as shown in Figure 3.2. Access in and out of that box is made by way of communication mechanisms defined in the module header definition, which may consist of **interaction points** and **exported variables**. These are the only means of communication the instance has with the enclosing environment defined by the parent module instance.

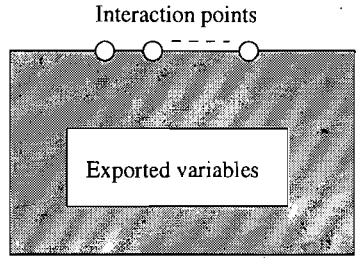


Figure 3.2: **The 'black box' module instance concept.** A module instance is created from a module header and body pair of a generic module. The only means of communication with a module instance are the **external interaction points** and **exported variables** defined in that module header. The communication mechanisms are described in Section 3.6.

From this point forward we refer to a module instance simply as an "instance", and a generic module definition as a "module".

3.3 Module body definition

A module body defines the internal behavior of a module instance created with that body. This behavior is characterized in terms of an extended finite state machine, as defined in Section 2.4.

At any discrete point in time, the extended state of this automaton is described by the current:

- control state;

- values of local variables;
- set of children instances;
- communication link structure of the module instance's interaction points and its children instances' interaction points;
- the contents of the queues associated to the instance's interaction points.

A module body definition is made up of three parts. The **initialization part** defines the *initial states* of the module instance. The **transition part** defines the *next-state relation* of the local EFSM. The **declaration part** declares local variables. These different parts are discussed in detail below.

An instance created from an associated module body with an empty **transition part** has no internal behavior, and is called *inactive*. An instance with a non-empty **transition part** is called *active*. An inactive instance, once initialized, performs no action. An active instance acts in a supervisory capacity over its children instances, and may dynamically change the descendant module instance hierarchy and communication link structure.

3.3.1 Declaration part

The declaration part of a module body contains the usual Pascal declarations of **types**, **constants**, **variables**, **procedures** and **functions**, and the following Estelle-specific objects:

- States and state sets;
- Module headers and associated module bodies;
- Module variables;
- Channels;
- Internal interaction points.

The **state** declaration enumerates the finite set of control states of the local automaton. A **state set** declares a single identifier which refers to a collection of control states.

Module headers and associated **module bodies** declare children generic modules of the containing module body definition.

A **module variable** serves as a reference to a module instance of a certain generic module definition. A **module variable** is declared to be of a certain **module header** type, and is initialized with an associated **module body**.

Channels and **internal interaction points** define communication mechanisms between the module and its children modules. They are described in Section 3.6.

3.3.2 Initialization part

The initialization part defines the *initial state* of a module instance. It is declared as a special transition in a module body called the *initialize transition*, which is fired immediately on initialization of a module variable with that associated body. The *initial state*:

- defines the initial descendant instance hierarchy and communication link structure;
- places the local EFSM in an initial control state;
- sets the initial values of some local variables.

The initialization statements for the instance structure and communication link structure and local variables are contained in the **action part** of the initialize transition. The initial control state is specified in the transition **to-clause**. These elements of a transition are described below.

More than one *initial state* may be defined by the *initialize transition*. The actual state initialized may be selected non-deterministically or according to the value of a **module header** parameter.

3.3.3 Transition part

The transition part defines a finite set of transitions which constitute the *next-state relation* of the local automaton.

A transition has four separate parts as shown in Figure 3.3:

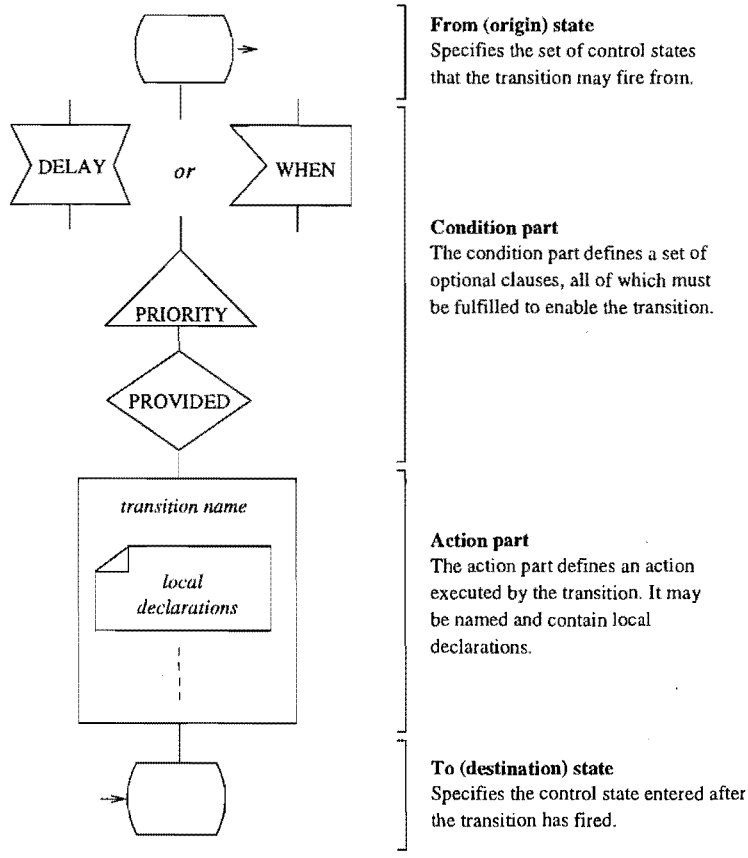


Figure 3.3: **An Estelle/GR transition.** An Estelle transition specifies an origin and destination state pair, optional conditions, and action statements as separate symbols in a flow diagram.

The **'from' clause** specifies the finite set of control states from which the transition may fire. It is treated as an enabling condition and included in the condition part.

The **condition part** defines a set of optional enabling conditions called *clauses*, all of which must be satisfied for the transition to be able to fire, or "enabled".

The conditions implied by each clause are as follows:

- A **when-clause** requires a specified message (or "interaction") at the front of a specified interaction point queue.
- A **provided-clause** specifies a boolean condition which must evaluate to **true**.
- A **priority-clause** specifies a transition execution priority.

In addition, a **delay-clause** specifies minimum and maximum times a transition must remain enabled before it may fire.

Each condition clause may appear at most once in a single transition definition. An omitted condition clause is considered to be fulfilled. A transition with an empty condition part (including no **from state** clause) is permanently enabled. Note that a **delay clause** and a **when clause** may not appear in the same transition definition.

The **action part** defines an optionally named action to be executed by the transition. This action may change the current module instance state, and output interactions to the environment. The action is defined as a sequence of Pascal and Estelle statements which are executed *atomically*. Each transition may declare local data and data manipulation functions.

Estelle defines the following statements which may be included in addition to Pascal statements in the action part:

- **Output** outputs an interaction through an interaction point.
- **Attach** and **detach** respectively create and destroy an *attach-segment*.
- **Connect** and **disconnect** respectively create and destroy a *connect-segment*.
- **Init** creates a module instance with an associated body.

- **Terminate** and **release** destroy a module instance.
- **All** is a non-deterministic repetition operator over a defined domain.
- **Forone** is a non-deterministic selector operator over a defined domain.
- **Exist** is a non-deterministic boolean operator over a defined domain.

The **'to' state** specifies the control state to be entered after the transition has fired.

3.4 Hierarchical structuring of modules

An Estelle system has a static module hierarchy which may non-deterministically result in a number of different instance hierarchies during execution of the system. Estelle is therefore said to have a *dynamic structure.*

The definition of a module body may contain further module definitions, which are *children* or *descendant* modules of that module body. Each descendant module body may in turn contain further module definitions, resulting in the **static module hierarchy** of an Estelle system. The root of this hierarchy is the *specification module*, which specifies global attributes of the system. Figure 3.4 is an example of a static module hierarchy.

A *module instance hierarchy* describes the dynamic hierarchy of components during the execution of a system. A number of different instance hierarchies are possible when a generic module definition consists of more than one associated module body. For example, Figure 3.5 illustrates two potential instance hierarchies which may arise from the single static module hierarchy of Figure 3.4 due to creation of the instance `Module A` with alternative body definitions.

The use of different associated bodies in Figure 3.4(a) and (b) results in significant differences in complexity in the two instance hierarchies. Figure 3.4(a) has two more instances than 3.4(b), as well as an extra level of depth in the hierarchy. If `Sub-module 1` and/or `Sub-module 2` in turn had alternative bodies, substantial variations in the instance hierarchy would be possible from the single static module hierarchy.

It is also important to stress the ability of a module instance to dynamically create and release or terminate any of its descendant instances. Thus during execution, an Estelle

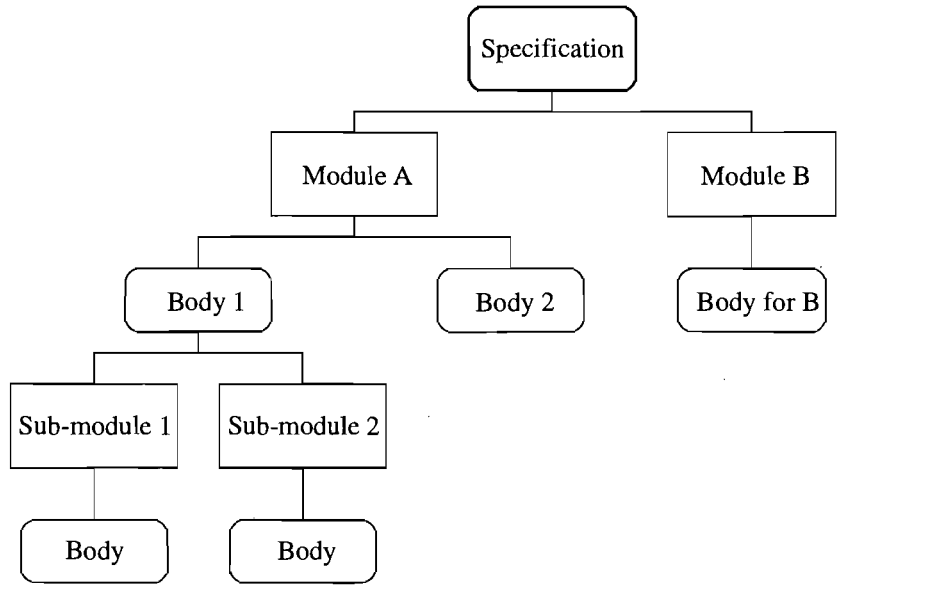
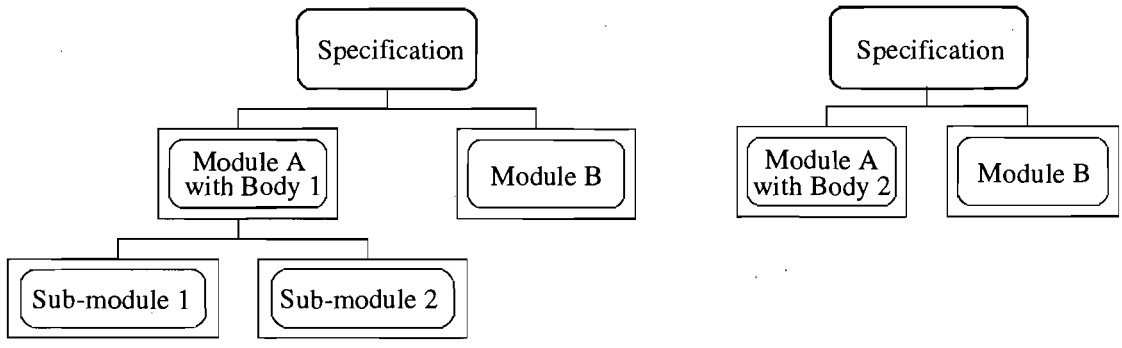


Figure 3.4: **A static hierarchy of generic module definitions.** In the hierarchy, children of a module body define descendant modules; children of a module header define associated module bodies of the generic module.



(a) Module A instantiated with Body 1 (b) Module A instantiated with Body 2

Figure 3.5: **Tree representation of module instance hierarchies .** These two instance hierarchies may result during the non-deterministic execution of an the static Estelle hierarchy defined in Figure 3.4. In (a), a child instance of `Specification` has been created from the pairing of `Module A` and its associated `Body 1`. In (b) the instance has been created with the associated `Body 2`.

system may move from the instance hierarchy of Figure 3.4(a) to that of 3.4(b) in a single discrete step.

Figure 3.6 shows an alternative but equivalent visualization of the module instance hierarchies of Figure 3.6, using a 'nested box' notation. In this notation, an instance is shown as a single block and descendent instances are graphically included in the parent block. This notation is also useful for showing the communication link structure, as described in Section 3.6.

3.5 Concurrency between modules

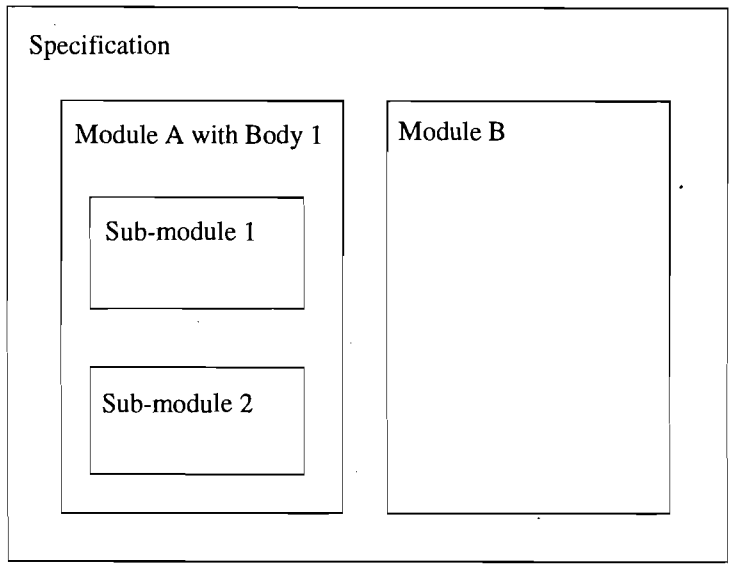
An Estelle instance hierarchy defines a fixed configuration of one or more independently-executing systems. Each system may be substructured into instances which execute either synchronously in parallel or in a non-deterministic way with sibling instances, supervised by the parent instance. There can be no concurrency between ancestor and descendant instances.

This concurrency between instances in the hierarchy is strictly determined by:

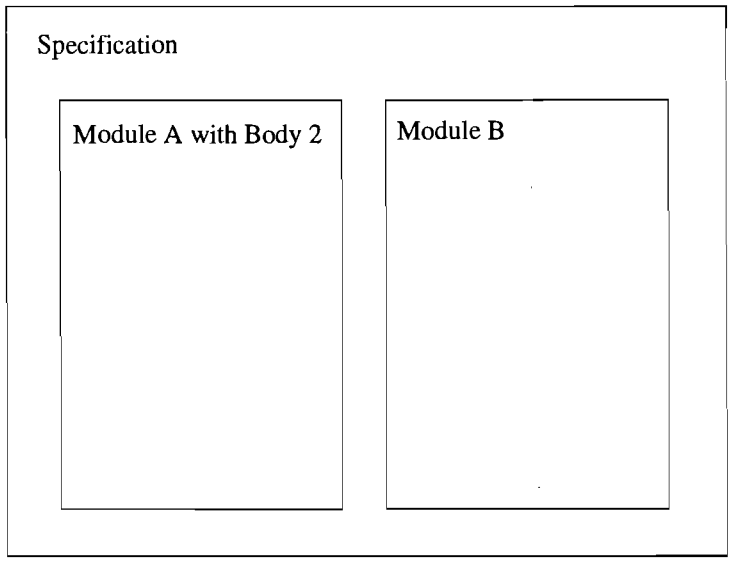
- their position in the instance hierarchy;
- the Estelle principle of execution priority for a transition of a parent instance over transitions of its children;
- the *structural attributes* of ancestor instances.

Transitions of a parent instance have priority over those of its children instances. While a module instance transition is fired, the execution of all children instance transitions is suspended. This priority relation is transitive, which excludes any concurrency amongst instances in an ancestor/descendant relationship.

The priority principle means that an active instance acts in a supervising capacity over its children instances. Intuitively, when a transition of a child instance is *enabled*, it must first "ask" its parent for permission to fire. This permission can only be granted after all transitions that were previously given permission to fire have completed. If transitions of multiple children are simultaneously enabled, the selection of which transition(s) are granted permission to fire depends on the parent instance's *structural attribute*.



(a) Module A instantiated with Body 1



(b) Module A instantiated with Body 2

Figure 3.6: **Nested box representation of module instance hierarchies .** In a module instance hierarchy, each instance has a fixed descendant module hierarchy and communication link structure (i.e. one body definition), making the nested box representation possible.

The structural attribute of an active instance is defined in the generic module header definition, and may take the values of **systemactivity**, **systemprocess**, **activity** or **process**. In the following, we refer to an attributed module by its attribute, e.g. an **activity module** is a module defined with the **activity** attribute. The hierarchical definition of modules must follow the following five rules:

1. Each active module must be attributed.
2. A system module (i.e. **systemactivity** or **systemprocess** module) cannot be a descendent of an attributed module.
3. **Process** and **activity** modules must be descendants of a system module.
4. **Process** or **systemprocess** modules may only be substructured into **process** or **activity** modules.
5. **Activity** or **systemactivity** modules may only be sustructured into **activity** modules.

From these rules, it follows that system modules may not be included within an *active* module, and are therefore not 'supervised' by a parent instance. Thus system modules execute *asynchronously in parallel.*

(System)process and **(system)activity** modules supervise different forms of concurrency between their children instances. Children instances of a **(system)process** execute *synchronously in parallel*; children instances of a **(system)activity** execute in a *nondeterministic* way. Intuitively, the **(system)process** module instance grants an enabled transition of each child instance permission to fire; a **(system)activity** module instance only grants permission to one randomly-selected transition out of those offered by its children instances.

3.6 Inter-module communication

A module instance is able to communicate with other instances in an Estelle system by two mechanisms, defined in the module header:

- *Message exchange* through **interaction points.**
- *Restricted sharing of variables* by way of **exported variables.**

Message exchange is the primary form of communication in Estelle, and may take place between arbitrary module instances between which a communication link has been created. Sharing of variables is an elegant method of communication between a module and its parent instance.

3.6.1 Message exchange

Estelle module instances exchange messages, or "interactions", as an extension of their internal EFSM. Interactions output by this EFSM are sent to another instance by way of dynamically linked interaction points. Interactions received at an interaction point are stored in unbounded FIFO queues to be consumed as inputs by the EFSM.

An Estelle system has a dynamic communication link structure, which may change during the specification execution. The permissible dynamic configurations are limited by the static definition of **channels** and **interaction points**. See Figure 3.7. Actual links are created dynamically between interaction points of the same channel.

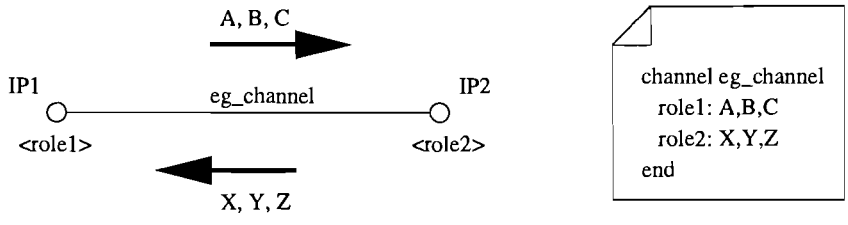


Figure 3.7: **Channels, roles and interaction points.** A channel defines two roles which may be assumed by interaction points at opposite ends of a communication link. Each role defines the finite set of interactions which may be sent by an interaction point of that role.

An **interaction point** is an abstract, bi-directional interface through which a module may send and receive interactions. *External* interaction points, defined as part of the externally-visible module header, exchange interactions with the external environment. *Internal* interaction points on the other hand, are defined as part of the module body, and exchange interactions with child modules of that body (an alternative to shared variables).

A **channel** defines two communication **roles** which may be assumed by interaction points. Each role defines a finite set of interactions, which establishes the set of interactions that an interaction point of that role may send. The two roles of a channel act in opposite directions across a communication link, thus the opposite role implicitly defines the finite set of interactions receivable at an interaction point.

All interaction points in a communication link must reference the same channel, therefore this definition must occur in a common ancestor instance of all these interaction point definitions.

Each interaction point also specifies either an *individual* or a *common* queuing model for received interactions. All interaction points of a module instance which use the common queuing model share a common queue, otherwise an interaction point has its own dedicated queue.

A communication link in Estelle is dynamically composed from a number of communication *segments*, each of which is defined in exactly one module instance. Estelle defines two types of segments differentiated by their purpose: An *attach-segment* defines a portion of a link over which interactions are forwarded. A *connect-segment* defines a portion of a link over which interactions are exchanged. A properly-defined communication link consists of exactly one *connection-segment* and zero or more *attach-segments.*

A communication segment is defined between two interaction points. Two segments which share an interaction point define part of a single link. A module instance may dynamically modify the interaction points between which a segment is defined, which changes the communication link structure. The rules by which communication segments may be composed to form a communication link are described below.

Figure 3.8 shows a communication link established between interaction points `A` and `C` of module instances `X` and `Z`, respectively. The link is composed of two segments: an *attach-segment* from `A` to `B`, and a *connect-segment* from `B` to `C`. The common interaction `B` joins the two segments to form a longer segment. `B` is unaware of the interactions it forwards, and cannot send any interactions itself.

The interaction points all share a common channel declaration, which therefore must occur in instance `W` or one of its ancestors.

An *attach-segment* is created by the **attach** operation, and binds an external interaction

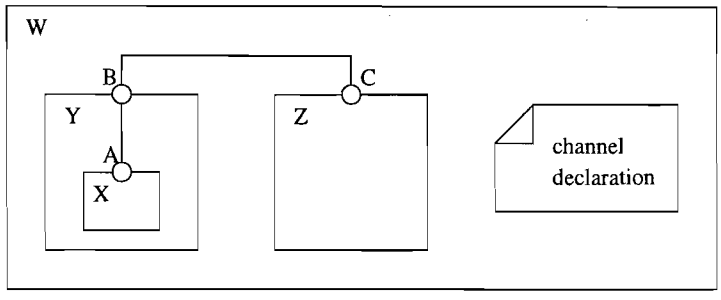


Figure 3.8: **An established communication link.** The communication link AC between interaction points A and C of modules X and Z is composed of two segments: an attach-segment AB and a connect-segment BC. Interaction points A and C *exchange* interactions; B *forwards* all interactions received.

point of the current instance to an external interaction point of a child instance. These interaction points share a common role, and the link allows the child instance to forward and receive messages to/from the external environment of its parent instance.

A *connect-segment* is created by the **connect** operation, and binds two interaction points visible within a single instance. The interaction points may be external interaction points of child instances, or internal interaction points of the parent instance. Connected interaction points have opposite channel roles, and exchange messages.

The Estelle communication link model does not permit multi-cast communication within a communication link. A link may only contain a single *connect-segment*, and an *attach-segment* may only be composed with another *attach-segment* in a parent or a child instance. Only two segments may share an interaction point.

3.6.2 Sharing of variables

A module instance is able to grant read and write access to a local variable for its parent instance by declaring a variable as *exported* in the module header definition.

Race conditions for access to a shared variable are precluded by the extension of the parent's priority over its child instances, as described in Section 3.5 above.

Parent and child instances may of course communicate by message exchange, but a shared variable is often more elegant.

3.7 Time in Estelle

The semantics of Estelle are defined independently of time, as this is seen to be implementation-dependent. Transitions are atomic and instantaneous, regardless of the internal complexity of a transition action part.

The protocol designer may however introduce a timing construct in the form of a transition **delay clause** to model relative execution times of transitions. A delay on a transition requires that the transition remain enabled for a specified amount of time before it may fire. If two transitions have identical condition parts except for their delay values, the transition with the shorter delay will fire (or from another point of view "complete" a semantically time-dependent activity) first.

A semantic indication of the unit of time measurement may be set in the specification module. This measurement is consistent throughout the specification.

Chapter 4

Existing visualizations of Estelle

To date, no formal, standard graphical representation for Estelle has been proposed. Nor have any tools been developed which allow graphical definition of a general Estelle system.

As far as the author is aware, only two tool exists which attempt to graphically represent an Estelle system in any way. These tools are described below, with emphasis on the graphical syntaxes which they employ. Section 4.3 summarizes the significance of these graphical syntaxes to the Estelle/GR.

4.1 GROPE

GROPE is a workstation tool for the Graphical Representation Of Protocols in Estelle (i.e. G.R.O.P.E.), developed at the University of Delaware. GROPE graphically animates the simulated execution of an Estelle protocol, by providing a graphical front-end for the WISE [oST92] Estelle interpreter.

Neither GROPE nor the WISE interpreter are still supported.

GROPE uses a *descriptive* notation to graphically represent the module instance hierarchy and state machines of an executing Estelle specification. The hierarchy is described in the familiar nested box notation and state machines are shown as state-transition diagrams. GROPE shows dynamic execution by animating the firing of transitions and passing of interactions over channels. It also provides user-controlled suppression of detail, including abstraction of complex state-transition diagrams.

This introduction to GROPE and its graphical syntax is taken mostly from [NP93]. All figures used in this section are reproduced from this article.

4.1.1 System structure description

Figure 4.1 shows the GROPE diagram of the module instance hierarchy of an example Alternating Bit Protocol.

Module and specification instances are shown by named rectangles, and the graphical inclusion of one rectangle within another describes a parent-child relationship. GROPE shows all hierarchical levels of the module hierarchy in a single diagram bounded by the specification instance border.

The border used to draw an instance rectangle differentiates between system and non-system instances. The thicker border of the `AlternatingBitExample` instance marks the specification instance as a system (specifically a systemprocess), while the thinner borders of the nested rectangles indicate non-system child instances. This detail is repeated in the parenthesized structural attribute following each instance name. The border does not differentiate between *process* and *activity* instances classes.

An interaction point is shown by a labeled dot (" ●"), and communication links are shown as curved or straight lines between two interaction points. GROPE labels the communication link with the associated channel name of the two linked interaction points, but gives no indication of their respective roles, or the interactions sendable by each.

GROPE differentiates between *external* and *internal* interaction points by positioning the former on the rectangle border of an instance, and allowing the latter to float freely within the instance rectangle. All of the interaction points in Figure 4.1 are *external.*

An interaction queue is shown by a "⊔", and GROPE differentiates between common and individual queues by graphically placing the former next to an instance name, and the latter next to an interaction point. As a descriptive notation, GROPE also shows the contents of these queues. Stacked dashes within a "⊔" indicate the number of queued interactions. For example, the individual queue of interaction point `N` of the `AlternatingBit[2]` instance contains three queued interactions.

GROPE animates the sending of an interaction between two interaction points by moving a dot over the communication link between them and adding a dash to the receiving queue.

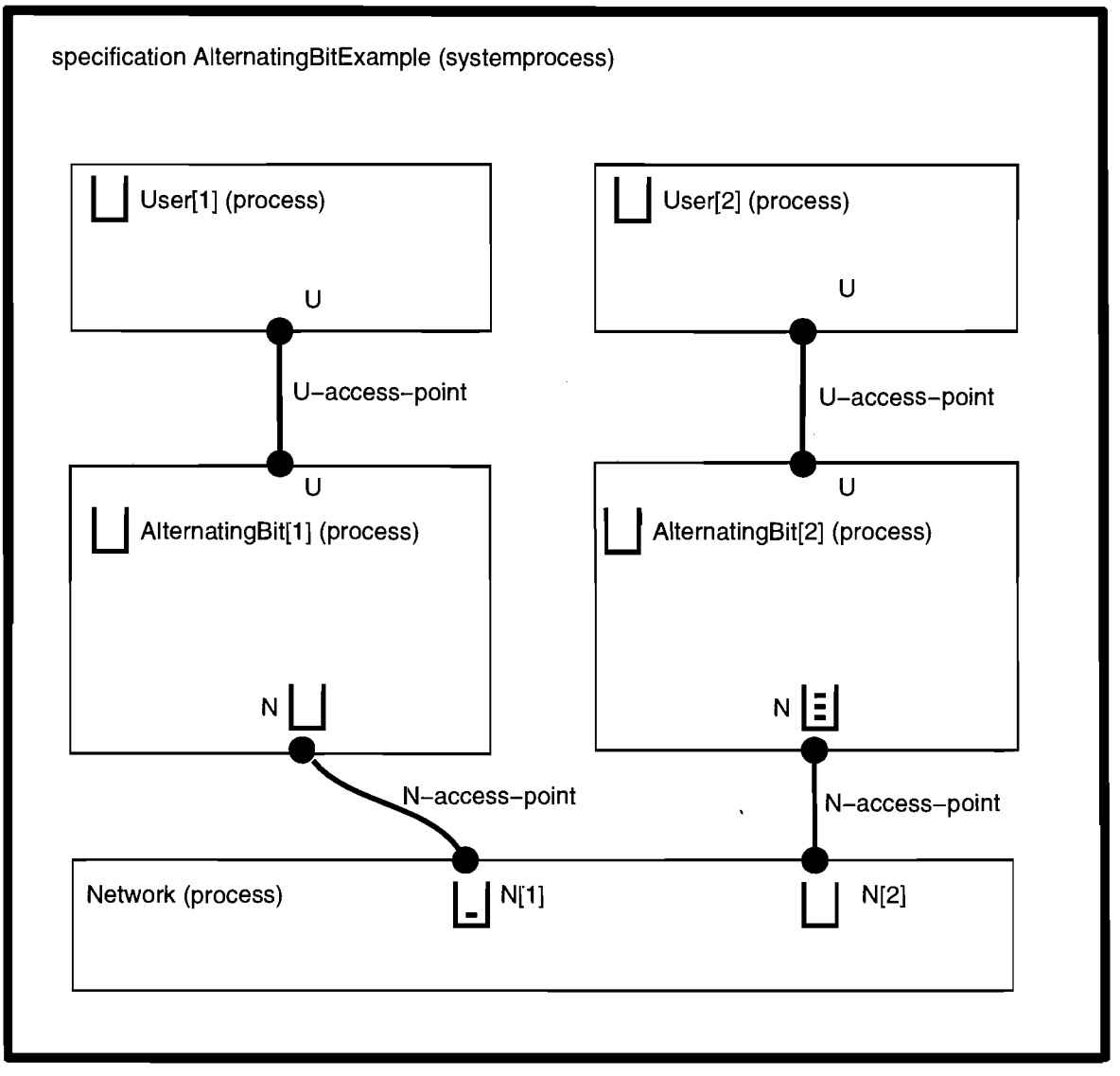


Figure 4.1: **GROPE: A system structure diagram.** GROPE uses a nested box notation to describe system structure. GROPE shows the entire depth of a system instance hierarchical in a single diagram.

4.1.2 Module behavior description

GROPE describes the module body internal behavior definition as a state-transition diagram. GROPE adds dynamic state information by always highlighting the current state of the EFSM, and animating the firing of a transition by moving a dot across a transition arc. Figure 4.2 shows a GROPE state-transition diagram in state ESTAB, and Figure 4.3 shows the animated firing of transition trans1.

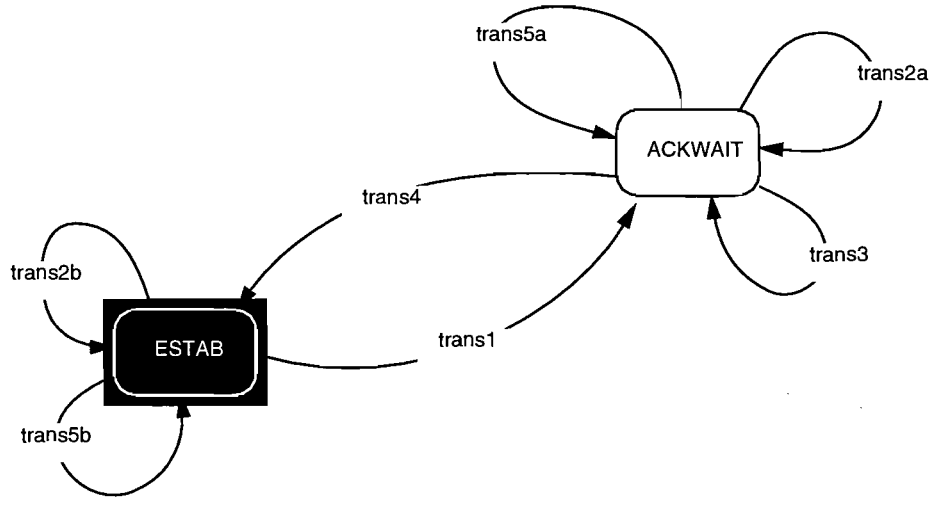


Figure 4.2: **GROPE: A state-transition diagram.** In the diagram the highlighted state ESTAB is the current state. Note that the details of transition definitions are not shown.

A GROPE state-transition diagram does not display any transition details, such as its condition and action parts.

GROPE provides abstractions for the simplification of a complex state-transition diagram with many states and/or transitions. A *macro-state* is a single state which represents a collection of states and all transitions between those states; and a *macro-transition* is a single transition which represents all transitions sharing the same origin and destination (macro)states. A macro-state is displayed as an oval with a double line, and a macro-transition as a boldened, unnamed arc. Figure 4.5 shows these constructs in use.

Consider the fairly complex state-transition diagram of Figure 4.4. Figure 4.5 shows the same diagram with states a, b and c grouped together as macro-state Mabc, states d, e, f

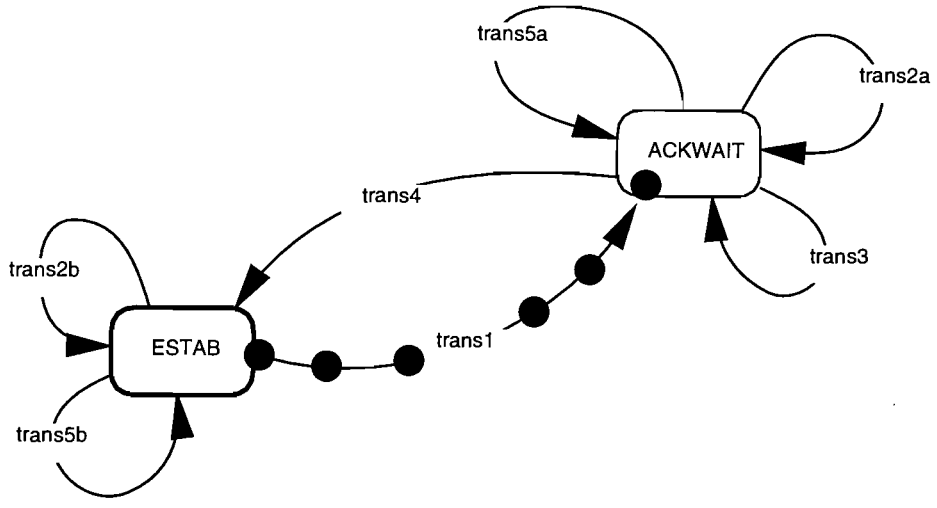


Figure 4.3: **GROPE: Animation of a transition firing.** In the diagram `trans1` is firing.

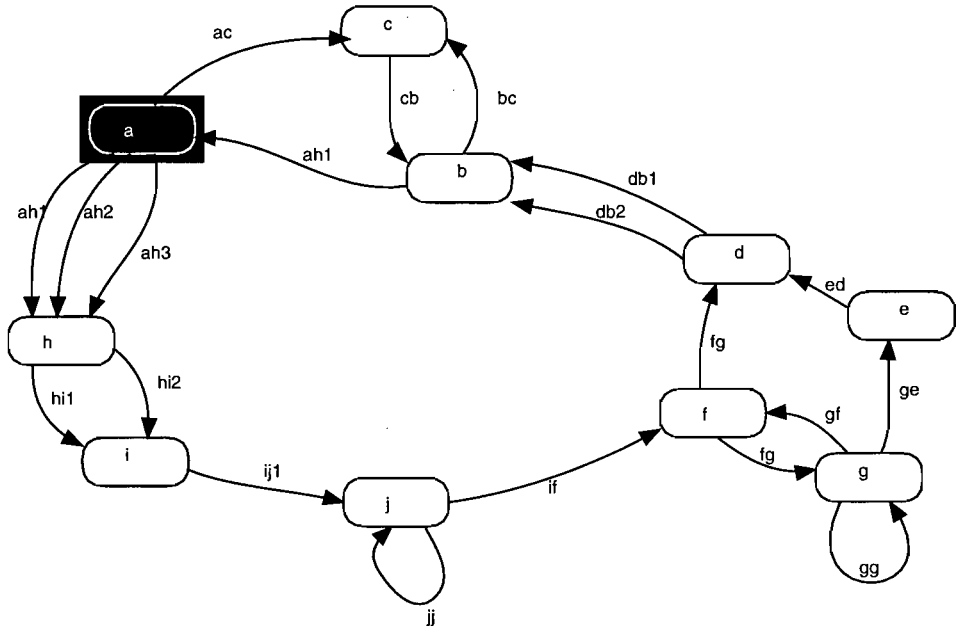


Figure 4.4: **GROPE: A complex state-transition diagram.** Figure 4.5 shows the result of GROPE abstraction techniques on this automaton.

and g grouped into macro-state Mdefg, and states h and i grouped into macro-state Mhi. Note that transitions between states grouped into a macro-state are also removed.

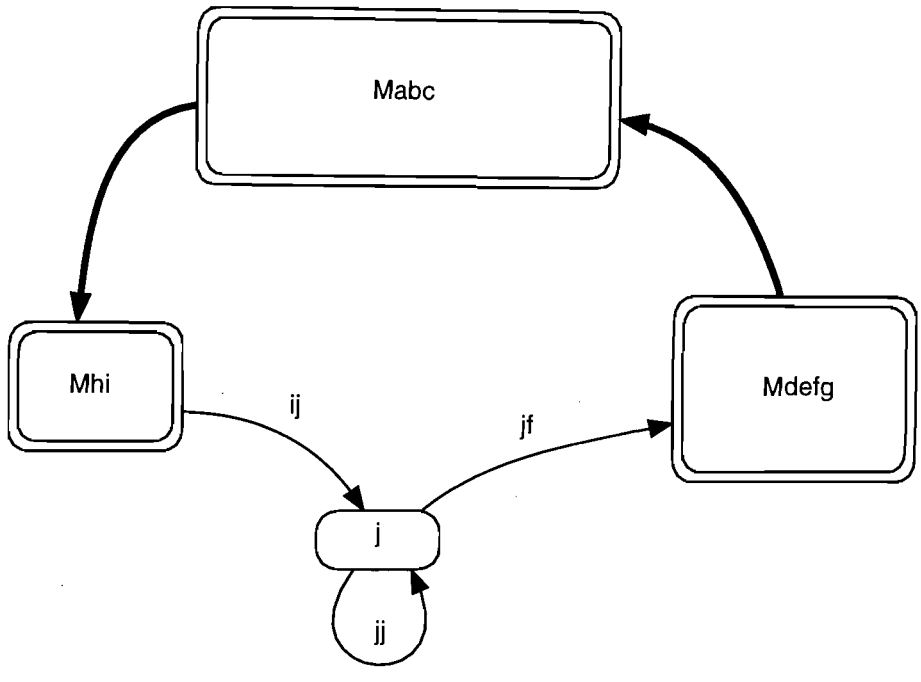


Figure 4.5: **GROPE: A simplified state-transition diagram.** The diagram shows a simplified view of the state-transition diagram of Figure 4.4. *Macro-states* and *macro-transitions* have been used to reduce the level of detail.

In addition, transitions ah1, ah2 and ah3 are grouped as a macro-transition between Mabc and Mhi, and transitions db1 and db2 are grouped into a macro-transition between Mdefg and Mabc.

By the introduction of macro-states and macro-transitions, Figure 4.4 has been reduced from ten states and twenty transitions to four states and five transitions. User-controlled grouping of states and transitions into macro-states and macro-transitions enables abstraction and simplification of a complex state-transition diagram.

4.2 ASA+

ASA+ is a software tool for designing and analyzing a system defined in its own modified version of Estelle called **LSA+**. ASA+ consists of a graphical editor, a simulator and a verification kernel. In this section we discuss the graphical editor **saedit**, and the syntax it uses. More information on ASA+ may be found in [Sab97].

LSA+ extends Estelle with a rendezvous mechanism, greater sharing of variables, and probabilistic aspects, but removes all structural dynamism. A system defined by LSA+ has a static module hierarchy and communication structure, and only leaf modules of the hierarchy can have a defined behavior.

saedit supports the graphical definition of a module hierarchy using the Structured Analysis and Design Technique (SADT) formalism. Internal module behavior is still defined using a textual syntax.

4.2.1 System structure definition

saedit defines a system structure in the form hierarchical tree of modules, as shown in the screenshot of the editor in Figure 4.6.

Each node in the tree defines a module, and child nodes define descendant modules. A small cross in the lower right-hand corner of a module indicates that it is *active*, i.e. has a defined behavior.

A single level of the substructure is defined using the SADT formalism, as shown in Figure 4.7. This is based on a different structural model to that of Estelle and we do not discuss it further.

4.3 Significance to the Estelle/GR

The different uses of the graphical syntaxes of the Estelle/GR, GROPE and ASA+ are summarized as follows:

- **Estelle/GR** is a graphical syntax used to *define* a general ISO Estelle protocol.

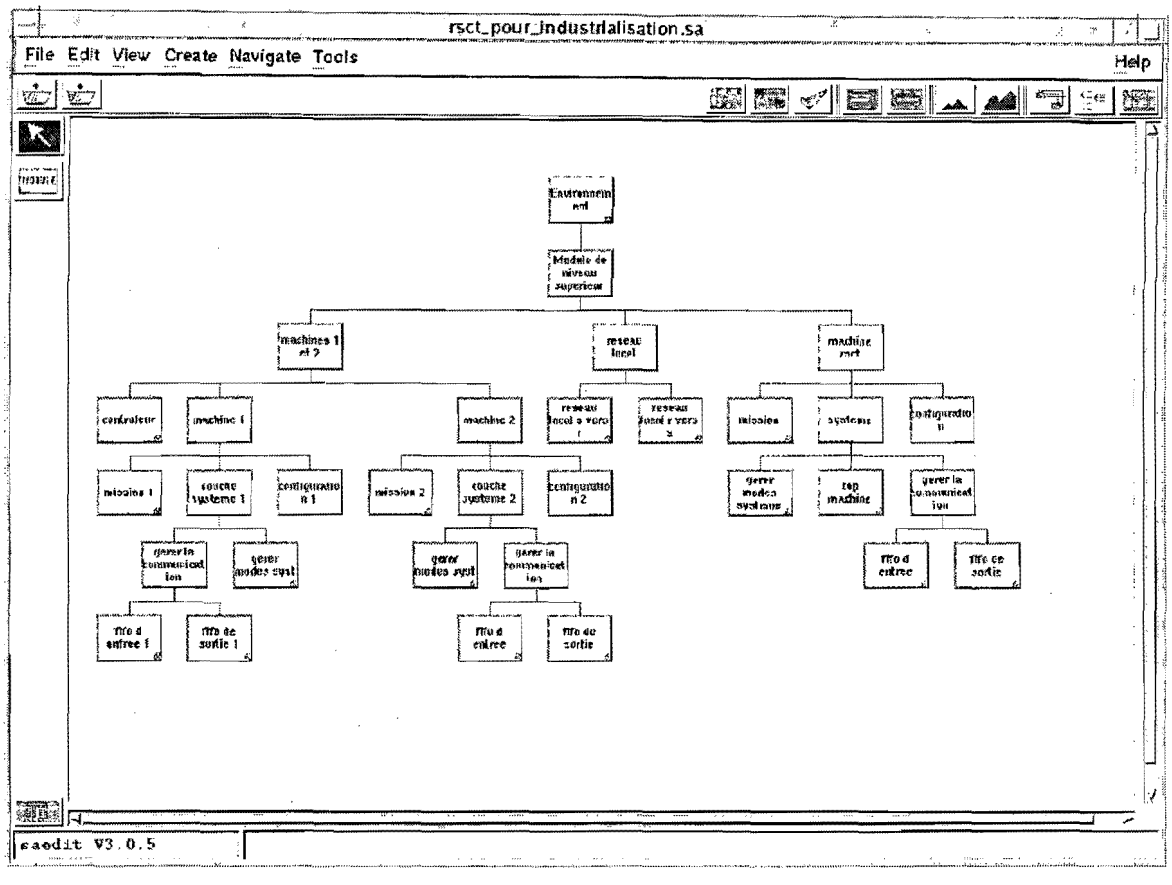


Figure 4.6: **ASA+: Definition of a module hierarchy.** The hierarchy is shown as a tree structure.

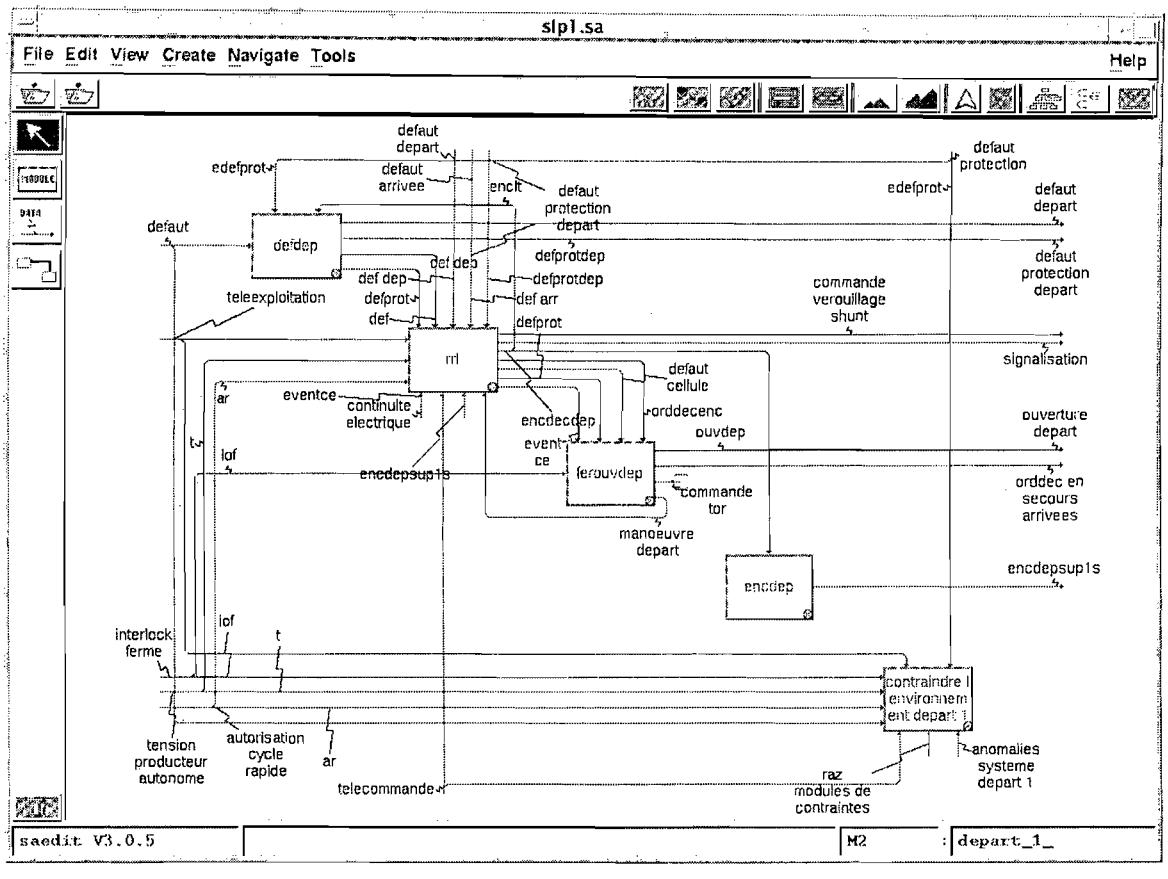


Figure 4.7: **ASA+: Definition of a module communication structure.** ASA+ uses a nested box notation to define the child modules and communication structure within a single module of the hierarchy. Only one level of the hierarchy is shown.

- **GROPE** uses a graphical syntax to *describe* dynamic protocol state during simulated execution.
- **ASA+** uses a graphical syntax to *define* the system structure of a non-ISO compliant version of Estelle.

These differences mean that very little of either GROPE or ASA+ is applicable to the Estelle/GR.

GROPE is an informal descriptive notation which both includes dynamic state information extraneous to a static definition, and excludes vital defining detail. It does however present many useful ideas for graphically representing an Estelle system.

We make use of the nested box notation for defining the structure of a static system in the **instance block diagram**. GROPE also introduces some useful finite state machine abstraction techniques which are used by the Estelle editor to present a simplified view of a complex internal module body behavior definition.

ASA+ does use a definitive notation for the structure definition, but the Estelle variant which it defines is substantially different to ISO Estelle. The Estelle/GR does use a similar structure tree notation extended to handle generic module definitions.

Chapter 5

The Estelle graphical representation

5.1 Introduction

This chapter defines the graphical syntax and graphical semantics of the Estelle/GR and forms the main body of this dissertation.

The Estelle/GR defines a graphical technique which can be used as an alternative or in conjunction to the textual Estelle/PR. This graphical language is fully capable of defining any specification expressible in the Estelle/PR as defined in the ISO Estelle standard [ISO97], and draws the following advantages from its graphical nature:

- An Estelle/GR diagram is *better structured* than an Estelle/PR text, as there is separation of system structure and behavior definition.
- The Estelle/GR syntax is *more abstract* than the Pascal-based Estelle/PR, which means that it is easier to understand and quicker to learn.

The Estelle/GR has received input from many sources in order to make it as acceptable a standard as possible. The representation was developed in cooperation with the Institut National des Télécommunications in France, which develops and maintains the most widely

used Estelle toolkit, XEdt [INT97] and has a significant influence on Estelle conventions, including system visualization.

The Estelle/GR was also submitted for comment to the global Estelle user community through the Estelle user mailing list (*e-mail:estelle@cs.umb.edu*). Many helpful suggestions were received and incorporated into the development of the Estelle/GR.

In addition, two existing tools which produce some visualization of an Estelle system were analyzed for suggestions on visualization conventions. **GROPE** and **ASA+** are presented in Chapter 4.

The Estelle/GR defines three separate diagrams which are used to define a system structure and individual module behaviors separately:

- A **structure tree diagram** defines a static generic module hierarchy. It can define the dynamic structure of any general Estelle system.
- An **instance block diagram** defines the initial instance hierarchy and communication structure of an Estelle system which has a statically-determinable initial state. It is used to define Estelle systems with a static structure.
- A **module body diagram** defines the internal behavior of either a module body defined in the structure tree diagram, or an instance defined in the instance block diagram.

A complete Estelle/GR specification diagram contains either a structure tree diagram or instance block diagram, and a module body diagram for each module body defined in the structure.

A specification diagram which defines system structure by way of a structure tree may also use a module instance diagram to define the initial state of the system.

In addition, the prototype editor described in Chapter 6 informally defines a **simplified automaton notation** which compliments the module body diagram. This notation, which is based on the state-transition diagram simplification techniques of GROPE, can be used to overview a large module body definition.

Each of the three major diagrams is presented in the following form:

- An example of the full diagram is presented.
- The graphical syntax and semantics of each grammatical element of the diagram are defined.

Note that the Estelle/GR is subject to the Estelle model semantics defined in the Estelle standard document [ISO97] and discussed in Chapter 3. For brevity we do not repeat these semantics in the following definitions, limiting our discussion to *graphical* meaning. The relationship between the Estelle/GR and Estelle/PR syntaxes is defined, which can be used to apply the formal semantics of the latter to the former.

5.2 Conventions used in defining the Estelle/GR

The Estelle/GR is defined using the textual meta-language defined in the Estelle standard (see [ISO97] § 6.1) extended by some graphical meta-symbols similar to those used in the SDL/GR grammar (see [ITU93b] § 1.5.3).

Non-terminals in the grammar are written in **bold** characters when discussed in the text. A non-terminal with the suffix **"-symbol"** produces a graphical symbol, and is called a *graphical non-terminal.*

Some of the syntax of the Estelle/PR is re-used in the Estelle/GR. To clearly distinguish new Estelle/GR non-terminals from existing Estelle/PR non-terminals, we prefix the new with **"GR-"** and the existing with **"PR-"**. As an example, the meaning of **PR-identifier** as used in the Estelle/GR syntax is defined as **identifier** in the Estelle/PR [ISO97]. Note that non-terminals of the Estelle/PR are never redefined, only re-used.

The meta-language used is based on Backus Naur Form, modified to permit greater convenience of description. Table 5.1 lists the meanings of the textual meta-symbols in the language.

In addition we define the following meta-symbols to describe the graphical syntax:

- contains
- is followed by

Metasymbol	Meaning
=	shall be defined to be
|	alternatively
.	end of definition
[x]	0 or 1 instance of x
{x}	0 or more instances of x
+{x}	1 or more instance(s) of x
(...)	grouping
"xyz"	terminal symbol xyz

Table 5.1: Estelle/PR meta-language symbols

- is connected to
- is associated with

Each of these is an infix operator having a graphical non-terminal as a left hand argument. The right hand argument is either a group of syntactic elements or a single syntactic element. If the right hand side of a production rule has a graphical non-terminal as its first element and contains one or more graphical operators, then that first graphical non-terminal is the left hand argument for each operator.

For example, the syntax definition

GR-body-declaration =
    GR-body-symbol
    CONTAINS
        PR-body-identifier
    IS ASSOCIATED WITH
        PR-interaction-point-declaration-part .

means that **GR-body-symbol** graphically contains **GR-body-identifier**, *and* **GR-body-symbol** is associated with **PR-interaction-point-declaration-part**.

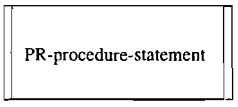
The meta-symbol **contains** indicates that its right-hand argument should be placed within its left-hand argument. For example:

GR-procedure-reference =

GR-procedure-reference-symbol
CONTAINS
PR-procedure-statement .

GR-procedure-reference-symbol =  .

is interpreted as:



The meta-symbol **is followed by** means that its right-hand argument follows (both logically and in drawing) its left-hand argument. If the right-hand argument is a group of syntactic elements, then each element follows from the left-hand argument separately. The left-hand argument is graphically joined to each of the elements of the right-hand argument by a *solid flow line* (see Section 5.3.1). If the right-hand argument produces to null, then the flow line is omitted.

The meta-symbol **is connected to** means that its right-hand argument is logically connected to its left-hand argument. Graphically the arguments are joined by a *dashed flow line* (see Section 5.3.1).

The meta-symbol **is associated with** means that the right-hand argument is a logical element of the left-hand argument. Graphically, the right-hand argument must be closer to the left-hand argument than any other graphical object.

5.3 Drawing rules

5.3.1 Flow lines

Flow lines connect logically associated objects in the graphical syntax. Flow lines can be dashed or solid.

Dashed flow lines are used to connect comments (see Section 5.3.2) to their symbols.

Solid flow lines are drawn between the left- and right-hand arguments of the **is followed by** meta-symbol. Solid flow lines have no directional marking: By convention, progression is from the top to the bottom of the diagram.

For neatness, flow lines are drawn on the perpendicular.

5.3.2 Comments

5.3.2.1 Syntax

GR-comment =
    GR-comment-symbol
    CONTAINS
        GR-comment-text
    CONNECTED TO
        [GR-commentable-symbol] .

GR-comment-symbol =  .

GR-comment-text = < *text* > .

GR-commentable-symbol =
    GR-external-body-symbol
    | GR-condition-symbol
    | GR-from-state-symbol
    | GR-from-state-set-symbol
    | GR-to-state-symbol
    | GR-any-symbol
    | GR-delay-symbol
    | GR-when-symbol
    | GR-priority-symbol
    | GR-procedure-reference-symbol
    | GR-output-symbol
    | GR-text-symbol
    | GR-attach-symbol

| GR-detach-symbol
| GR-connect-symbol
| GR-disconnect-symbol
| GR-initialize-symbol
| GR-release-symbol
| GR-terminate-symbol .

5.3.2.2 Graphical semantics

A **GR-comment** may be connected to any of the **GR-commentable-symbols** by a *dashed flow line* (see Section 5.3.1). The comment is interpreted as being associated to the object connected to.

Alternatively, a **GR-comment** may be contained within the enclosing border of a **GR-structure-definition-part**, **GR-body-definition-part**, **GR-child-instance** or **GR-transition-action-part**. The comment is interpreted as being associated to the object defined by the closest-containing frame.

The comment does not affect the formal interpretation of the specification in any way.

5.3.3 Diagram partitioning

A complex specification diagram is unlikely to fit on a single page. The Estelle/GR divides a specification diagram in a set of modular parts, partitioned along the following guidelines:

- The structure definition should be separate from the module body definitions.
- Each body definition should be isolated and uniquely identified by the structure tree path (see Section 5.3.4) for that body declaration.
- For added clarity, a comment should be associated with each body definition and the structure definition.

In general a special **GR-frame-symbol** is used to partition the diagram by enclosing each modular part of the diagram in a solid rectangle.

GR-frame-symbol =  .

5.3.4 Unique structure tree path

A structure tree path uniquely identifies a module body within a hierarchy of generic module definitions.

The path string is constructed by concatenating the names of headers and bodies lying on the direct path between the specification root and the body declaration, starting with the specification name and ending with that of the body being identified. The individual identifiers are separated with the "/" symbol to form a UNIX-like directory path.

For example, the unique structure tree path of the body for `Sub-module 1` in Figure 3.4 on Page 23 would be `/Specification/ModuleA/Body1/Sub_module1/Body` (the identifiers have been slightly modified to form proper Estelle identifiers). The structure tree path for the specification module would simply be `/Specification`.

The structure tree path for a module body declared in an instance block diagram is slightly different, as there is no separation of module header and module body. The path is constructed from the module instance names, terminating at the instance whose body is being identified.

As an example, the structure tree path for instance `Sub-module 1` in Figure 3.5 on Page 23 would be `/Specification/ModuleA/Sub_module1`. The specification instance path remains unchanged as `/Specification`.

5.4 Structure of an Estelle/GR specification diagram

An Estelle specification diagram consists of a single structure diagram, and a module body definition for each body declared in the structure diagram.

The start symbol of the Estelle/GR grammar is **GR-specification-diagram**.

5.4.1 Syntax

GR-specification-diagram =

GR-structure-definition-part
{ GR-body-definition-part } .

GR-structure-definition-part =
GR-frame-symbol
CONTAINS (
GR-structure-definition
[GR-specification-comment]) .

GR-structure-definition =
GR-structure-tree-diagram
| GR-instance-block-diagram .

GR-specification-comment =
GR-comment .

5.4.2 Graphical semantics

The structure diagram may take the form of either a structure tree or an instance block diagram. A **GR-structure-tree-diagram** can be used to define any general system structure.

A **GR-instance-block diagram** defines the *initial state* of an Estelle system which has a single initial state. A **GR-instance-block diagram** is useful for defining systems where generic modules have exactly one body and the system structure is *static*, as explained in Section 3.4.

A **GR-body-definition-part** defines the internal behavior of a module body declared in the structure definition. Each **GR-body-definition-part** is labeled with the unique *structure tree path* (defined in Section 5.3.4) of the module body it defines.

The **GR-specification-comment** is a specification-associated comment which floats within the enclosing **GR-frame-symbol** of the **GR-structure-definition-part**. It is intended for information such as the specification author, date, a brief synopsis, etc.

5.5 Structure tree diagram

A structure tree diagram defines the generic module hierarchy of an Estelle system as described in Section 3.4. Module header and body definitions are defined as nodes in the tree, and annotated with their structural attributes.

Header nodes are indicated by different rectangular symbols which define the mode of concurrency between their descendent modules, i.e. defines the header structural attribute. A header node also defines the external interfaces of a module by way of textual interaction point and exported variable declarations associated with the header node.

A body node is indicated by a rounded rectangle symbol. It defines internal interaction points in the same way as external interaction points are declared at a header node. A shaded body node indicates that the body is "external" and not defined in the specification diagram; an unshaded body is defined by a module body diagram.

The root node of the tree defines the specification module, and sets the specification-wide timescale and default queuing model options.

The structure tree does not show channel definitions (these are included in the module body definitions), and as it defines the statically-specified elements of the system structure, is unable to show communication links.

Figure 5.1 presents the structure tree diagram of the TFTP protocol from Appendix A.

5.5.1 Structure tree diagram

A **GR-structure-tree-diagram** defines a hierarchy of generic modules in which module headers and module bodies are separate nodes in a tree. The individual nodes are annotated with attributes relevant to the system structure.

5.5.1.1 Syntax

GR-structure-tree-diagram =
    GR-specification-module
    IS FOLLOWED BY
        { GR-child-module } .

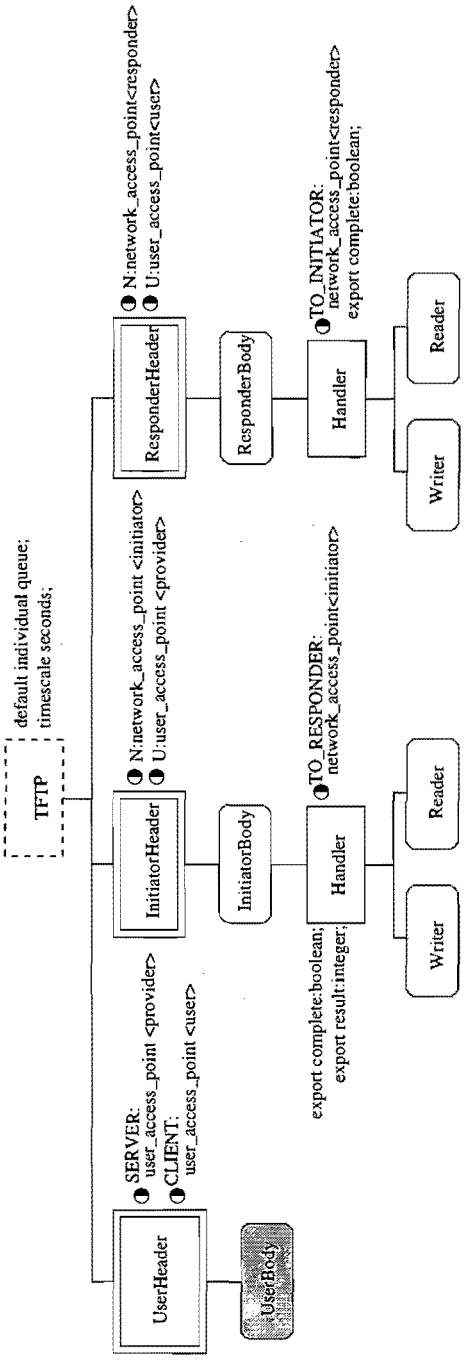


Figure 5.1: **Estelle/GR: A structure tree diagram.** The diagram shows the complete structure tree of the TFTP in Appendix A.

GR-specification-module =
GR-specification-symbol
CONTAINS
PR-specification-identifier
IS ASSOCIATED WITH (
[PR-default-options]
[PR-time-options]) .

GR-specification-symbol =
GR-systemactivity-symbol
| GR-systemprocess-symbol
| GR-unattributed-symbol .

GR-systemactivity-symbol =  .

GR-systemprocess-symbol =  .

GR-unattributed-symbol =  .

GR-child-module =
GR-module-header
IS FOLLOWED BY
+{ GR-module-body } .

5.5.1.2 Graphical semantics

The **GR-specification-module**, represented by a **GR-specification-symbol**, is the root node of the **GR-structure-tree-diagram** and defines the Estelle/PR specification module. The **GR-specification-symbol** defines the specification module system class which determines the mode of concurrency between **GR-child-modules**.

A **GR-child-module** defines the module header and associated module body parts of a descendent module. Recursive definition of **GR-child-modules** from **GR-module-body**

declarations defines the module hierarchy.

The internal behavior of the **GR-specification-module** or a **GR-module-body** is defined elsewhere in the **GR-specification-diagram** by a **GR-body-definition-part**.

5.5.2 Module header definition

A **GR-module-header** defines the module header part of a **GR-child-module**.

5.5.2.1 Syntax

GR-module-header =
GR-header-symbol
CONTAINS
PR-header-identifier [PR-parameter-list]
IS ASSOCIATED WITH
{ (GR-individual-queue-symbol
| GR-common-queue-symbol
| GR-default-queue-symbol)
GR-ip-identifier [GR-ip-dimensions] ":"
GR-channel-identifier "<" GR-role-identifier ">" }
IS ASSOCIATED WITH
["export" + PR-exported-variable-declaration] .

GR-header-symbol =
GR-systemprocess-symbol
| GR-systemactivity-symbol
| GR-process-symbol
| GR-activity-symbol .

GR-process-symbol =  .

GR-activity-symbol =  .

GR-common-queue-symbol = .

GR-individual-queue-symbol = ● .

GR-default-queue-symbol = ◑ .

GR-ip-dimensions = "[" PR-index-type-list "]" .

GR-channel-identifier = PR-identifier .

GR-role-identfier = PR-identifier .

5.5.2.2 Graphical semantics

A **GR-header-symbol** in the **GR-structure-tree-diagram** defines a descendent module of its parent module body defined by a **GR-specification-symbol** or **GR-body-symbol** in the structure tree.

A **GR-module-header** is represented in the structure tree by a **GR-header-symbol**. The symbol used defines the structural attribute of the header module. The choice of symbol is limited by the module structuring rules defined in Section 3.4

An interaction point declaration associated to a **GR-module-header-symbol** defines an *external* interaction point for the **GR-module-header**. Each interaction point is shown by a small circle which defines the queue model of that interaction. A solid circle defines an individual queue, an open circle a common queue, and a half-shaded circle specifies that the default queue option specified in the **GR-specification-module** is used.

Exported variables of the module header are declared in Estelle/PR syntax and associated with the **GR-header-symbol**. Parameters of the module header are declared as in Estelle/PR syntax as a list following the module header name.

5.5.3 Module body declaration

A **GR-module-body** declares an associated module body for a **GR-module-header**.

5.5.3.1 Syntax

GR-module-body =
(GR-body-declaration
IS FOLLOWED BY
{ GR-child-module })
| GR-external-body .

GR-body-declaration =
GR-body-symbol
CONTAINS
PR-body-identifier
IS ASSOCIATED WITH
PR-interaction-point-declaration-part .

GR-external-body =
GR-external-body-symbol
CONTAINS
PR-body-identifier .

GR-body-symbol =  .

GR-external-body-symbol =  .

5.5.3.2 Graphical semantics

A **GR-body-symbol** or **GR-external-body-symbol** in the **GR-structure-tree-diagram** defines an associated module body for the module header defined by its parent **GR-header-symbol** in the tree. The **GR-module-body** is defined in the parent module body of the **GR-module-header** which it follows in the structure tree.

A **GR-child-module** which follows a **GR-body-symbol** defines a descendent module for the **GR-module-body**.

An interaction point declaration associated to a **GR-body-symbol** defines an *internal* interaction point for this body.

The internal behavior for the module body is defined elsewhere in the **GR-specification-diagram**, in a **GR-body-definition-part** labeled with the structure-tree path (see Section 5.3.4) of the **GR-module-body**. This definition is conceptually included in the structure at the point of declaration of the module body.

Use of the **GR-external-body-symbol** explicitly states that this module body is *not* defined in the **GR-specification-diagram**. The **GR-external-body-symbol** has no associated interaction point declarations, may not define descendent modules, and has no **GR-body-definition-part**.

5.6 Instance block diagram

An instance block diagram defines the initial module instance hierarchy and communication structure of an Estelle system which has a statically-determinable initial state. This diagram is particularly useful for defining an Estelle system with a *static structure*, i.e. one that does not change from the initial state.

The diagram defines both the static module and communication structure of the system and the module body initialization parts needed to instantiate the system to the initial state it represents.

The instance block diagram defines module instance information in a single rectangle which includes the definition information of the module header, module body and module variable required to create and initialize that instance. Graphical inclusion of one instance within another defines the instance hierarchy.

The diagram also includes full communication link information. Channel declarations are included textually within an instance rectangle, and interaction points are shown as 'dots' either on an instance rectangle border (external) or floating within the rectangle (internal). Links are shown as solid lines between interaction points.

Figure 5.2 reproduces the instance block diagram for the TFTP example presented in Appendix A. This system has a dynamic structure and therefore the diagram is not definitive of the system structure and only shows the system in its initial state.

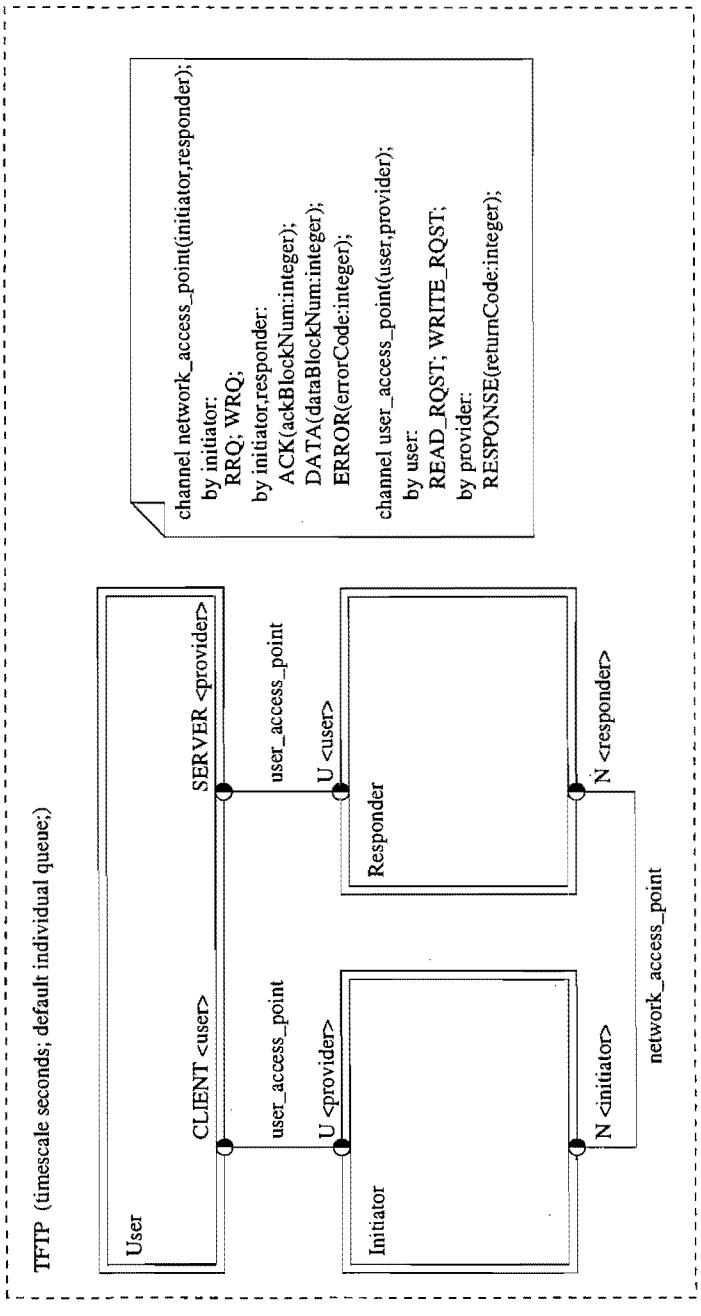


Figure 5.2: **Estelle/GR: An instance block diagram.** The diagram shows the initial system structure of the TFTP from Appendix A. Compare to Figure 5.1 on Page 51: The TFTP has a dynamic structure in which the **Header** modules are not part of the initial state. Therefore this notation cannot be used to define this system structure.

5.6.1 Instance block diagram

5.6.1.1 Syntax

GR-instance-block-diagram = GR-specification-instance .

GR-specification-instance =
    GR-specification-symbol
    IS ASSOCIATED WITH
        PR-specification-identifier
        [PR-default-options]
        [PR-time-options])
    CONTAINS (
        [GR-specification-comment]
        GR-instance-substructure) .

GR-specification-symbol =
    GR-systemactivity-symbol
    | GR-systemprocess-symbol
    | GR-unattributed-symbol .

GR-specification-comment =
    GR-comment .

5.6.1.2 Graphical semantics

The **GR-specification-symbol** defines the outer boundaries of a specified system. Associated to it is the specification name (**GR-specification-identifier**) and the values of the specification-wide default queue (**PR-default-options**) and timescale (**PR-time-options**) options.

The choice of **GR-specification-symbol** defines the mode of concurrency between child instances defined in the **GR-instance-substructure**. The **GR-module-block-diagram** uses the same symbols as the **GR-structure-tree-diagram** (see Section 5.5.1), enlarged

to frame the entire **GR-specification-instance** definition.

The optional **GR-specification-comment** is a specification-related comment.

The **GR-instance-substructure** defines the descendent instance hierarchy and communication link structure.

The internal behavior of the **GR-specification-instance** is defined in a **GR-body-definition-part** elsewhere in the **GR-specification-diagram**.

5.6.2 Instance substructure

A **GR-instance-substructure** defines the descendent instance hierarchy and communication link substructure of the **GR-specification-instance** or the **GR-child-instance** in which it is contained, and the initialization part of the module body part of this instance.

5.6.2.1 Syntax

GR-instance-substructure =
{ GR-child-instance }
[GR-channel-definition-part]
{ GR-attach-segment }
{ GR-connect-segment }
{ GR-ip-declaration } .

GR-channel-definition-part =
GR-declaration-symbol
CONTAINS
PR-channel-definition .

GR-declaration-symbol =  .

The following Estelle/PR syntax channel definition is copied directly from [ISO97] § 7.3.4.1 for reference purposes.

PR-channel-definition = PR-channel-heading PR-channel-block .

PR-channel-heading = "channel" PR-channel-identifier "(" PR-role-list ")" ";" .

PR-channel-identifier = PR-identifier .

PR-role-list = PR-identifier "," PR-identifier .

5.6.2.2 Graphical semantics

A **GR-child-instance** defines a descendent instance of the containing **GR-specification-instance** or **GR-child-instance**. Recursive nesting of **GR-child-instances** defines the module instance hierarchy.

The other parts of a **GR-instance-substructure** define the communication link structure of the containing instance.

The **GR-channel-definition-part** defines new channels which can be used to define interaction points of this instance and nested **GR-child-instances**. Channel definitions use Estelle/PR syntax enclosed in a **GR-declaration-symbol**.

A **GR-ip-declaration** contained within a **GR-instance-substructure** defines an *internal* interaction point. An *external* interaction point is defined on the border of the enclosing **GR-specification-symbol** or **GR-header-symbol**. **GR-ip-declaration** syntax is defined below in Section 5.6.3.

A **GR-attach-segment** or **GR-connect-segment** defines an instantiated communication link between two interaction points. It is shown by a solid line drawn between two **GR-ip-symbols** and labeled with the name of the common channel which the interaction points refer to. as shown in Figure 5.3. The **PR-channel-identifier** identifies the channel to which the two interaction points are associated. The segment in Fig 5.3 defines two interaction points named `IP1` and `IP2` and associated to the channel `common_channel`. The interaction point declaration syntax is defined in Section 5.6.3 below.

A **GR-attach-segment** may be defined between an external interaction point of the enclosing module instance and an external interaction point of a **GR-child-instance**, which are associated to the same channel and roles.

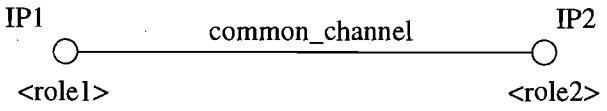


Figure 5.3: **Estelle/GR: A communication segment in the instance block diagram.** The label on the link specifies the channel shared by the interaction points.

A **GR-connect-segment** may be defined between any two 'connect' interaction points associated to the same channel but opposite roles, where a 'connect' interaction point is any internal interaction point defined in the **GR-instance-substructure** or any external interaction point of a **GR-child-instance**.

A **GR-attach-segment** or **GR-connect-segment** between two arrays of interaction points indicate a one-to-one in order linking of the individual interaction points in the arrays. Only arrays with the same absolute dimensions may be linked by a segment. Sub-ranges of an interaction point array with different segment structures must be declared as separate interaction points.

The instance substructure defines the following statements in the **GR-initialization-part** of the module body part of the containing **GR-child-instance**: Each **GR-child-instance** defines a **GR-initialize** statement; Each **GR-attach-segment** defines a **GR-attach** statement; Each **GR-connect-segment** defines a **GR-connect** statement.

5.6.3 Interaction point declaration

5.6.3.1 Syntax

GR-ip-declaration =
    GR-ip-symbol
    IS ASSOCIATED WITH
        PR-identifier [GR-ip-dimensions]
        "<" GR-role-identifier ">" .

GR-ip-symbol =
    GR-common-queue-symbol

| GR-individual-queue-symbol
| GR-default-queue-symbol .

5.6.3.2 Graphical semantics

External and internal interaction points are declared with the same syntax. They are differentiated by their placement in a **GR-instance-block-diagram**. *External* interaction points of the module header part are defined on the border of the enclosing **GR-header-symbol**. *Internal* interaction points of the module body part are contained within the symbol border (but outside the borders of descendent instance definitions).

An interaction point is defined by a small shaded or unshaded circle (the **GR-ip-symbol**) which specifies whether that interaction point has an individual or a common queue.

If **GR-ip-dimensions** are specified, an array of interaction points is defined.

The set of interactions which an interaction point can send is defined by the **GR-role-identifier** associated to the **GR-ip-symbol**. The **GR-role-identifier** must be selected from the **PR-role-list** of the channel associated to a **GR-connect-segment** or **GR-attach-segment** terminating at this interaction point.

5.6.4 Child instance

A **GR-child-instance** defines a descendent module instance of the containing **GR-module-instance** or **GR-specification-instance**.

5.6.4.1 Syntax

GR-child-instance =
    GR-header-symbol
    IS ASSOCIATED WITH (
        GR-instance-header-part
        { GR-ip-declaration })
    CONTAINS

[GR-instance-comment]
(GR-instance-substructure | "external") .

GR-instance-header-part =
GR-instance-identifier [GR-instance-dimensions]
[GR-instance-exported-variables]

GR-instance-identifier = PR-identifier .

GR-instance-dimensions = "[" PR-index-type-list "]" .

GR-instance-parameters = "(" PR-parameter-list ")" .

GR-instance-exported-variables = "export" +{ PR-exported-variable-declaration } .

GR-instance-comment = GR-comment .

5.6.4.2 Graphical semantics

The **GR-child-instance** declares and initializes a module variable identified by **GR-instance-identifier**, the module header from which the module variable is declared, and an associated module body with which it is initialized.

The instance definition is enclosed by a **GR-header-symbol**, indicating the extent of the definition. The **GR-header-symbol** defines the structural attribute of the module header which specifies the mode of concurrency between descendent module instances defined in the **GR-instance-substructure**.

A **GR-ip-declaration** defines an *external* interaction point of the module header, and is placed on the border of the **GR-header-symbol**.

The **GR-instance-exported-variables** defines exported variables of the module header part.

GR-instance-dimensions defines an array of module instances, which corresponds to a definition of a module variable array.

The use of the keyword "external" within a **GR-header-symbol** explicitly states that the internal structure and behavior of this instance is not defined in the **GR-specification-diagram**.

If not declared "external", the **GR-instance-substructure** defines the descendent instance hierarchy and communication link structure of the instance, and a **GR-body-definition-part** elsewhere in the **GR-specification-diagram** defines its internal behavior.

5.7 Module body diagram

A module body diagram defines the internal behavior of a module body as discussed in Section 3.3. The bulk of this is taken up by the transition part definition, which also includes the special initialize transition which defines the initialization part.

Transition are defined as trees, in which each graphical symbol defines a condition to be met, or statement to be executed. The conditions and statements are evaluated in order from top to bottom in the tree. A split in a tree defines multiple trees in which the conditions above the split apply to each branch.

To demonstrate how transition trees define the automaton of an extended finite state machine, Figure 5.4 shows a set of transition trees and Figure 5.5 shows the automaton which they define.

The first transition in Figure 5.4 is the initialize transition.

5.7.1 Body definition part

A **GR-body-definition-part** defines the internal behavior of an module body declared in the **GR-structure-definition**. This may be any of a **GR-module-body** or **GR-specification-module** declared in the **GR-structure-tree-diagram**, or a **GR-module-instance** or **GR-specification-instance** declared in the **GR-instance-block-diagram**.

5.7.1.1 Syntax

GR-body-definition-part =
    GR-frame-symbol

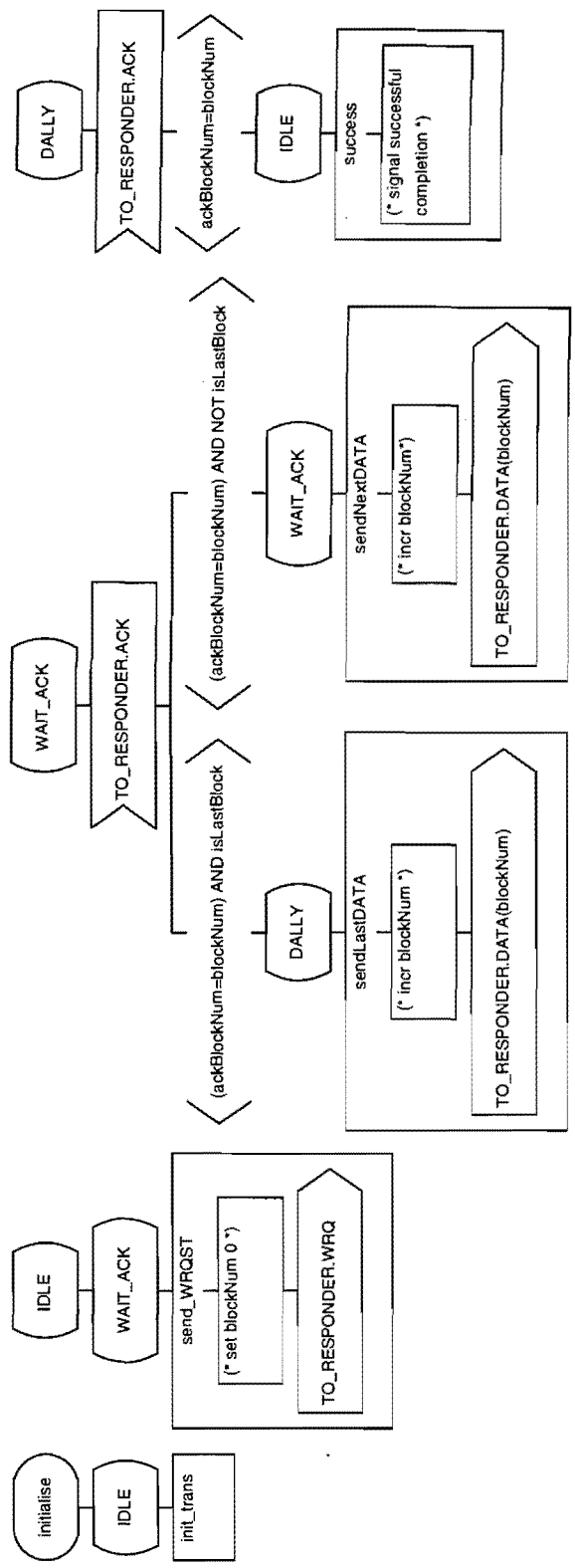


Figure 5.4: **Estelle/GR: Examples of transition trees defining module behavior.** The automaton defined by these transitions is shown in Figure 5.5.

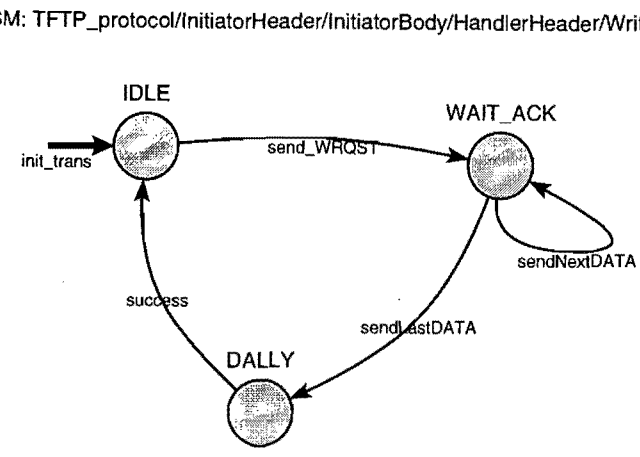


Figure 5.5: **Estelle/GR: An automaton defined by transition trees.** This automaton is produced from the transition trees defined in Figure 5.4.

IS ASSOCIATED WITH
GR-body-definition-label
CONTAINS (
GR-body-definition
[GR-body-comment]) .

GR-body-definition-label = < *text* > .

GR-body-definition =
[GR-declaration-part]
[GR-initialization-part]
[GR-transition-part] .

GR-body-comment = GR-comment .

5.7.1.2 Graphical semantics

A **GR-body-definition-part** is partitioned from other **GR-body-definition-parts** and the **GR-structure-definition-part** by an enclosing **GR-frame-symbol**. It is uniquely identified by the **GR-body-definition-label**, which has the value of the structure tree path (defined in 5.3.4) for the module body or instance whose internal behavior is being defined.

The body definition can be optionally commented using the **GR-body-comment**. This does not have any semantic meaning.

The actual behavior is specified by the **GR-body-definition** which has three separate parts which we define in the following sections.

5.7.2 Declaration part

The **GR-declaration-part** performs declarations pertaining to a module's internal behavior. When the system structure is defined by a **GR-structure-tree-diagram**, it may also contain structure-related declarations.

5.7.2.1 Syntax

GR-declaration-part =
    GR-declaration-symbol
    CONTAINS
        { GR-declarations } .

GR-declarations =
    PR-constant-definition-part
    | PR-type-definition-part
    | PR-state-definition-part
    | PR-state-set-definition-part
    | PR-variable-declaration-part
    | PR-procedure-and-function-declaration-part
    | PR-channel-definition

| PR-module-variable-declaration-part .

See [ISO97] § 7.3 for the Estelle/PR syntax of each of the above declarations.

5.7.2.2 Graphical semantics

If the system structure is static, i.e. the **GR-structure-definition** contains a **GR-instance-block-diagram** and not a **GR-structure-tree-diagram**, then the **GR-channel-definition** and **GR-module-variable-definition** are not permissible declarations, as these are explicitly declared in the instance block diagram.

Each of the permissible **GR-declarations** may occur in any order and each may occur more than once, except for the **PR-state-definition-part**.

These declarations have no graphical representation, and are defined in the textual Estelle/PR syntax.

5.7.3 Initialization part

A **GR-initialization-part** defines the *initialize transition* (see Section 3.3.2) of the containing **GR-module-definition-part**.

5.7.3.1 Syntax

GR-initialization-part =
    GR-initialization-symbol
    IS FOLLOWED BY
        GR-transition-definition .

GR-initialization-symbol = 

.

5.7.3.2 Graphical semantics

The initialize transition is defined as a transition of the **GR-transition-part** which starts with the special **GR-initialization-symbol**.

Only one initialize transition may exist per **GR-body-definition**.

See [ISO97] §7.5.10.2 for constraints on the **PR-initialization-part** which also apply to the **GR-initialization-part**.

5.7.4 Transition part

The **GR-transition-part** defines a set of transitions which define the EFSM of the containing **GR-module-definition-part**. Each transition is defined as a flow diagram of conditions and action statements which is read sequentially from top to bottom.

5.7.4.1 Syntax

GR-transition-part = { GR-transition-definition } .

GR-transition-definition =
    (GR-transition-condition-part | GR-transition-action-part) .

GR-transition-condition-part =
    GR-condition
    IS FOLLOWED BY
        +{ GR-transition-condition-part | GR-transition-action-part } .

GR-condition =
    GR-provided-clause
    | GR-from-clause
    | GR-to-clause
    | GR-delay-clause
    | GR-when-clause
    | GR-priority-clause
    | GR-any-clause .

GR-transition-action-part =
    GR-transition-statement-part
    IS FOLLOWED BY
        [GR-to-clause] .

GR-transition-statement-part =
    GR-frame-symbol
    CONTAINS (
        [GR-transition-name]
        [GR-transition-declaration-part]
        [GR-transition-comment]
        GR-statement-group) .

GR-transition-name = PR-identifier .

GR-transition-declaration-part =
    GR-declaration-symbol
    CONTAINS (
        PR-constant-definition-part
        PR-type-definition-part
        PR-variable-declaration-part
        PR-procedure-and-function-declaration-part) .

GR-transition-comment = GR-comment .

GR-statement-group =
    GR-statement
    IS FOLLOWED BY
        [GR-statement-group] .

GR-statement =
    GR-procedure-reference
    | GR-output
    | GR-pascal-code

| GR-initialize
| GR-terminate
| GR-release
| GR-attach
| GR-detach
| GR-connect
| GR-disconnect
| GR-all-statement
| GR-forone-statement .

5.7.4.2 Graphical semantics

The **GR-transition-part** consists of zero or more **GR-transition-definitions**, each of which defines a new **PR-transition-declaration**. If the **GR-transition-part** is empty, instances created with this body will be *inactive*. Otherwise the instance is *active*.

A **GR-transition-definition** has three distinct syntactic parts: the **GR-transition-condition-part** followed by the **GR-transition-statement-part**, followed (optionally) by the **GR-to-clause**. A properly-defined transition must contain a **GR-transition-statement-part**.

The **GR-transition-condition-part** defines a series of **GR-conditions** which must all evaluate to true to enable the transition. If a **GR-transition-condition-part** is not defined, then the transition is always enabled.

Each **GR-condition** is shown as a separate symbol in the **GR-transition-definition**. The clauses may appear in any order, but only once each. The graphical ordering of the clauses does not affect interpretation of the transition. An exception to this is the **GR-any-clause**. A **GR-any-clause** may occur multiply in a single **GR-transition-condition-part**, and its position is significant, as it introduces new identifiers which may be referenced by **GR-conditions** and **GR-statements** following it in the transition definition.

The **GR-transition-statement-part** is enclosed in a solid **GR-frame-symbol** to indicate atomic execution of the enclosed statements, and separate it from the clause symbols of

the **GR-transition-condition-part**. It is optionally named and commented, and may declare local variables and data manipulation functions in the contained **GR-transition-declaration-part**.

The **GR-statement-group** defines the individual statements to be executed. Each **GR-statement** is shown as a separate symbol in the **GR-transition-definition**, and may occur in any order any number of times.

The **GR-to-clause** may appear either as a clause in the **GR-transition-condition-part** or follow from the **GR-transition-action-part**, but not both in the a single **GR-transition-definition**.

5.7.4.3 Defining a transition tree

The above semantics do not define the ability within the syntax to split a **GR-transition-condition-part** into multiple branches, the equivalent to defining the *nested transitions* of [ISO97] §7.5.2.4.

A single **GR-condition** may in fact be followed by more than one **GR-transition-condition-part** or **GR-transition-action-part**, providing each branch starts with a **GR-transition-action-part** or the same **GR-condition**.

This creates a split in the **GR-transition-definition**, in which the **GR-conditions** defined before the split apply to every resultant branch. A branch can in turn be split into further branches, to form a complex *transition tree* structure. Each branch that does not contain a further split must contain a **GR-transition-action-part**.

A transition tree can always be expanded into a finite set of *simple transitions*. A simple transition is defined for each **GR-transition-action-part** found in the tree. The simple transition is composed of all the symbols on a direct path from the root of the transition tree to the **GR-transition-action-part** (or the **GR-to-clause** if that is defined to follow the action part).

Figure 5.6 shows an example of a complex transition tree and its expansion.

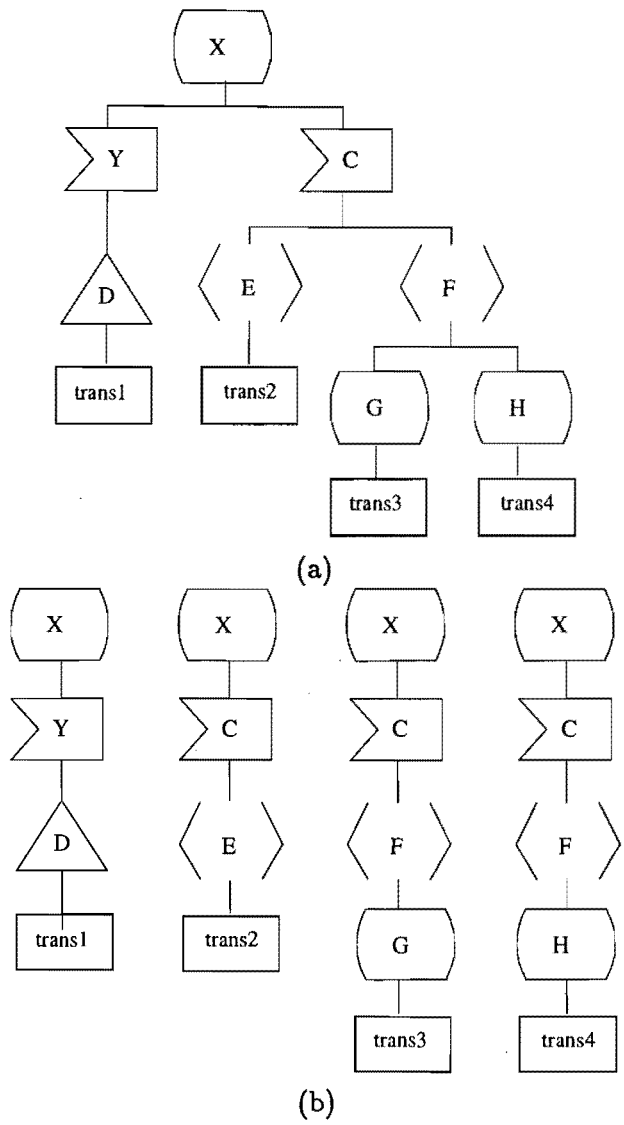


Figure 5.6: **Estelle/GR: A transition tree and its expansion into simple transitions.** The single branched transition tree in (a) defines four simple transitions, shown in (b).

5.8 Transition condition clauses

Each condition clause is defined by a specific Estelle/GR symbol which contains the specifics of that condition. According to the Estelle model, each condition symbol may only occur once in a *simple transition* tree.

5.8.1 Provided clause

The provided clause specifies a boolean condition which must evaluate to true for the transition to be able to fire.

5.8.1.1 Syntax

GR-provided-clause =
    GR-condition-symbol
    CONTAINS (
        PR-boolean-expression
        | "otherwise") .

GR-condition-symbol = ⟨ ⟩ .

5.8.1.2 Graphical semantics

A **GR-provided-clause** uses a flow diagram *decision* symbol to indicate a transition condition.

The ⟨ otherwise ⟩ form defines a branch in a transition tree to be executed in the event of no other **GR-provided-clause** being satisfied.

The ⟨ otherwise ⟩ form may only be defined as a sibling to another **GR-provided-clause** and may never start a **GR-transition-definition**.

5.8.2 From clause

The from clause specifies a finite set of control states (previously defined in the **GR-state-definition-part** of the containing **GR-body-definition**) from which a transition may be executed.

5.8.2.1 Syntax

GR-from-clause =
GR-from-symbol
CONTAINS
GR-from-expression .

GR-from-symbol = GR-from-state-symbol | GR-from-state-set-symbol .

GR-from-state-symbol =  .

GR-from-state-set-symbol =  .

GR-from-expression = GR-from-include-list | GR-from-exclude-list .

GR-from-include-list = +{ GR-from-include-element } .

GR-from-include element = GR-state-identifier | GR-state-set-identifier | "*" .

GR-from-exclude-list = "*(" +{ PR-from-exclude-element } ")" .

GR-from-exclude-element = GR-state-identifier | GR-state-set-identifier .

GR-state-identifier = PR-identifier .

GR-state-set-identifier = PR-identifier .

5.8.2.2 Graphical semantics

A **GR-from-clause** is indicated by an oval state symbol. Where more than one state is intended, a shadowed version of the state symbol is used. The **GR-from-state-symbol** is differentiated from the **GR-to-state-symbol** by an arrow exiting the state, whereas the latter shows an arrow entering the state.

A **GR-from-clause** defines an expression from which the set of states from which this transition may validly fire is calculated. The absence of a **GR-from-clause** in a transition is equivalent to declaring the transition as fireable from all control states in the module body.

A **GR-from-clause** is represented by either the **GR-from-state-symbol** or **GR-from-state-set-symbol**, dependent on whether the **GR-from-expression** evaluates to *one* or *many* states, respectively.

The **GR-from-expression** has two forms. The **GR-from-include-list** defines a set of states and state sets from which the transition may fire. The **GR-from-exclude-list** defines a set of all states and state sets from which the transition may *not* fire. In the **GR-from-exclude-list** form, the set of states from which the transition may validly fire consists of all states *excepting* those in the **GR-from-exclude-list**.

The special symbol "*" is a shorthand meaning *all states* defined in the module body.

5.8.3 To clause

The to clause specifies a single control state (previously defined in the **GR-state-definition-part** of the containing **GR-body-definition**) which the system enters after a transition fires.

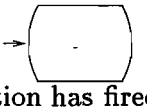
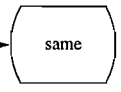
5.8.3.1 Syntax

GR-to-clause =
    GR-to-state-symbol
    CONTAINS
        GR-to-expression .

GR-to-state-symbol =  .

GR-to-expression =
GR-state-identifier
| "same"
| "-" .

5.8.3.2 Graphical semantics

If a **GR-transition-definition** does not contain a **GR-to-clause**, or if it uses one of the forms  or  is used, then the current state does not change after the transition has fired.

5.8.4 Delay clause

A delay clause defines a minimum time period for which an enabled transition must be kept waiting before firing, and a maximum time period it may be kept waiting before being guaranteed to fire.

5.8.4.1 Syntax

GR-delay-clause =
GR-delay-symbol
CONTAINS
GR-delay-min-wait-time
[GR-delay-max-wait-time | "*"] .

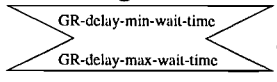
GR-delay-symbol =  .

GR-delay-min-wait-time = PR-expression .

GR-delay-max-wait-time = PR-expression .

5.8.4.2 Graphical semantics

The **delay-clause** is indicated by an hour-glass symbol to indicate the notion of time passing.

The **GR-delay-min-wait-time** and **GR-delay-max-wait-time** expressions are meant to be arranged one above the other in the hourglass symbol of the **GR-delay-clause**, as in 

.

The **GR-delay-min-wait-time** defines the minimum time a delayed transition will be forced to wait before being permitted to fire. **GR-delay-max-wait-limit** sets the maximum time it may be kept waiting before being guaranteed to fire.

The **GR-delay-max-wait-time** expression is optional, and 

is semantically equivalent to 

.

Using "*" in place of the **GR-delay-max-wait-time** expression signifies that there is no upper bound on the time a transition may be kept waiting, i.e. the transition may acceptably never fire.

5.8.5 When clause

A when clause requires the presence of a specified interaction at the front of a specified interaction point queue to enable a transition to fire.

5.8.5.1 Syntax

GR-when-clause =
GR-when-symbol
CONTAINS
PR-when-ip-reference .

GR-when-symbol =  .

5.8.5.2 Graphical semantics

The **GR-when-clause** is indicated by a flow-diagram *input* symbol.

Identifiers introduced as interaction arguments in the **PR-when-ip-reference** are visible to *all* of the **GR-conditions** and **GR-statements** in a **GR-transition-definition**, even those statements that may precede the **GR-when-clause** in the **GR-transition-condition-part**.

5.8.6 Priority clause

A priority clause sets a transition's priority such that, if more than one transition is simultaneously enabled, the transition with the higher priority will be selected to fire

5.8.6.1 Syntax

GR-priority-clause =
GR-priority-symbol
CONTAINS
PR-priority-constant .

GR-priority-symbol =  .

5.8.6.2 Graphical semantics

If the **GR-priority-clause** is absent then the lowest priority is assumed.

5.8.7 Any clause

An any clause is a shorthand for defining a set of transitions grouped by a domain.

5.8.7.1 Syntax

GR-any-clause =
GR-any-symbol
CONTAINS
PR-domain-list .

GR-any-symbol =  .

5.8.7.2 Graphical semantics

A **GR-any-clause** is drawn as a truncated and reversed funnel, to indicate the process of expansion.

Identifiers declared in the **PR-domain-list** are available to all **GR-conditions** and **GR-statements** following the **GR-any-symbol** in the **GR-transition-definition**.

A **GR-transition-definition** containing a **GR-any-clause** is a shorthand for a set of transitions. The expansion of this shorthand is described in [ISO97] § 7.5.2.4.2.

5.9 Transition action statements

Only Estelle-specific statements have dedicated Estelle/GR symbols. Pascal statements which may be used to define a transition action are included in a general **GR-pascal-code** symbol, or combined into a procedure definition. A procedure reference has a dedicated Estelle/GR symbol.

5.9.1 Procedure reference

A procedure invokes a Pascal procedure visible in the current scope.

5.9.1.1 Syntax

GR-procedure-reference =

GR-procedure-symbol
CONTAINS
PR-procedure-statement .

GR-procedure-symbol =  .

5.9.1.2 Graphical semantics

A **GR-procedure-reference** is indicated by an *SDL procedure call* symbol.

Use of a **GR-procedure-reference** is equivalent to performing the same **PR-procedure-statement** in a **GR-pascal-code** statement, but it exposes this semantic action more clearly.

5.9.2 Output

An output statement sends an (optionally) parameterized interaction through a specified interaction point.

5.9.2.1 Syntax

GR-output =
GR-output-symbol
CONTAINS
PR-interaction-reference [PR-actual-parameter-list] .

GR-output-symbol =  .

5.9.2.2 Graphical semantics

A **GR-output** is indicated by a flow diagram *send* symbol.

Use of a **GR-output** is equivalent to performing the same statement in a **GR-pascal-code** statement. The **GR-output** exposes this key action more clearly.

5.9.3 Pascal code

The Pascal code symbol defines a sequence of textual Pascal statements in the transition action definition.

5.9.3.1 Syntax

GR-pascal-code =
    GR-text-symbol
    CONTAINS
        PR-statement-sequence .

GR-text-symbol =  .

5.9.3.2 Graphical semantics

The **GR-text-symbol** can contain any Estelle/PR statements, including all the **GR-statements** defined in this section. In fact the **GR-statement-group** could be defined within a single **GR-pascal-code** statement. However, use of a separate symbol for these statements is recommended as this better exposes the Estelle-specific actions of a transition.

It is also recommended that if a **GR-pascal-code** segment contains a group of semantically-related statements, they be converted into a procedure with a meaningful name, and then called using a **GR-procedure-reference**.

5.9.4 Attach

An attach statement establishes an *attach-segment* of a communication link between an external interaction point of the current module and an external interaction point of a child module.

5.9.4.1 Syntax

GR-attach =

GR-attach-symbol
CONTAINS
"attach" PR-external-ip "to" PR-child-external-ip .

GR-attach-symbol = GR-frame-symbol .

5.9.5 Detach

A detach statement may either detach a specific communication segment from an external interaction point of a child module, or all current *attach-segments* of that child module.

5.9.5.1 Syntax

GR-detach =
GR-detach-symbol
CONTAINS
"detach" (PR-external-ip | PR-child-external-ip | PR-module-variable) .

GR-detach-symbol = GR-frame-symbol .

5.9.6 Connect

A connect statement creates a *connect-segment* between two *connect-interaction-points*, where a *connect-interaction point* is either an external interaction point of a child module, or an internal interaction point of the current module.

5.9.6.1 Syntax

GR-connect =
GR-connect-symbol
CONTAINS
"connect" PR-connect-ip "to" PR-connect-ip .

GR-connect-symbol = GR-frame-symbol .

5.9.7 Disconnect

A disconnect statement may either disconnect a specific *connect-segment* in the module sub-structure, or all current *connect-segments* of specific child module.

5.9.7.1 Syntax

GR-disconnect =
    GR-disconnect-symbol
    CONTAINS
        "disconnect" (PR-connect-ip | PR-module-variable) .

GR-disconnect-symbol = GR-frame-symbol .

5.9.8 Initialize

The initialize statement creates and initializes and a module instance pointed to by a module variable.

5.9.8.1 Syntax

GR-initialize =
    GR-initialize-symbol
    CONTAINS
        PR-module-variable
        "with" PR-body-identifier ["(" PR-actual-module-parameter-list ")"] .

GR-initialize-symbol =  .

5.9.8.2 Graphical semantics

A **GR-initialize** statement is indicated by an *SDL create* symbol.

5.9.9 Terminate and Release

Terminate and release statements destroy a child module instance pointed to by a module variable, as well as interactions stored in the interaction queues of the module. 'Release' saves the queued interactions in an attached module.

5.9.9.1 Syntax

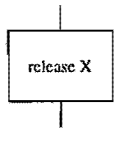
GR-terminate =
    GR-terminate-symbol
    CONTAINS
        "terminate" PR-module-variable .

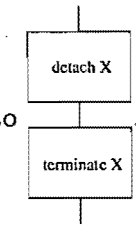
GR-release =
    GR-release-symbol
    CONTAINS
        "release" PR-module-variable .

GR-terminate-symbol = GR-frame-symbol .

GR-release-symbol = GR-frame-symbol .

5.9.9.2 Graphical semantics

Note that 

is equivalent to 

.

5.9.10 All statement

An all statement is a complex loop statement which executes a specified sequence of statements once for each element in a specified domain.

5.9.10.1 Syntax

We introduce a new graphical meta-symbol, **loops through** to describe a **GR-all-statement**. This operator means that the graphical end of the right-hand argument should be reconnected to the left hand argument, indicating a control flow loop.

GR-all-statement =
    GR-all-domain
    LOOPS THROUGH
        GR-statement-group .

GR-all-domain = "all" PR-domain-list .

5.9.10.2 Graphical semantics

The graphical structure of the **GR-all-statement** is shown in Figure 5.7.

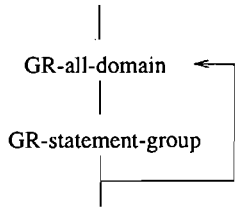


Figure 5.7: **Estelle/GR: An ALL statement.**

5.9.11 Forone statement

A forone statement is a non-deterministic decision statement which executes a specified statement sequence if a boolean expression evaluates true for at least one element in a domain. If the expression never evaluates to true, an alternative statement sequence is executed.

5.9.11.1 Syntax

GR-forone-statement =
    GR-forone-domain
    IS FOLLOWED BY (
        GR-forone-true-part
        GR-forone-otherwise-part) .

GR-forone-domain = "forone" PR-domain-list

GR-forone-true-part =
    GR-condition-symbol
    CONTAINS
        PR-boolean-expression
    IS FOLLOWED BY
        GR-true-statement-group .

GR-otherwise-statement-group = GR-statement-group .

GR-forone-otherwise-part =
    GR-condition-symbol
    CONTAINS
        "otherwise"
    IS FOLLOWED BY
        [GR-otherwise-statement-group].

GR-otherwise-statement-group = GR-statement-group .

5.9.11.2 Graphical semantics

The graphical structure of the **GR-forone-statement** is shown in Figure 5.8.

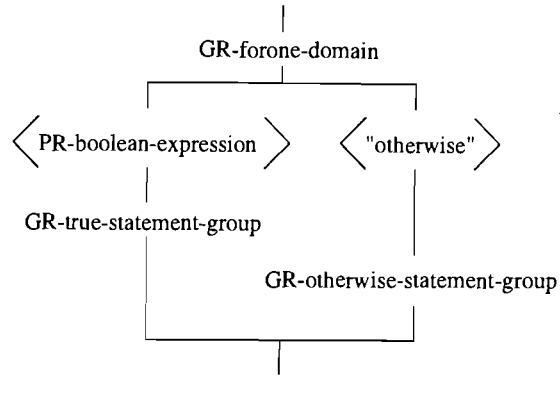


Figure 5.8: **Estelle/GR: A FORONE statement.**

If the **PR-boolean-expression** of the **GR-forone-true-part** evaluates to **true** for at least one of the elements of the **GR-forone-domain**, then the sequence of statements defined in **GR-true-statement-group** is executed. Otherwise the sequence of statements defined in the **GR-otherwise-statement-group** is executed

The **GR-forone-true-part** and **GR-forone-otherwise-part** splits the transition into two branches after the **GR-forone-domain** definition. This is temporary split, and the two **GR-statement-groups** are rejoined after their last **GR-statements**.

Chapter 6

A prototype graphical editor for Estelle/GR

6.1 Introduction

The editor transforms the Estelle/GR into a powerful graphical design technique by integrating this graphical form with existing text-based Estelle tools across a range of operating platforms.

The editor provides a structured interface for viewing and editing an Estelle specification in the graphical form defined by the Estelle/GR. The main functions of the editor, also illustrated in Figure 6.1, are as follows:

- *Import* - Import a general Estelle/PR document into the editor in Estelle/GR form.
- *Export* - Write an Estelle/GR document to a file in Estelle/PR form.
- *Document* - Generate postscript output of the graphical Estelle/GR diagrams.
- *Edit* - Provide syntax-directed editing of an Estelle/GR specification document.
- *Explore* - Provide structures and viewing mechanisms to quickly and easily be able to assimilate a specification's meaning.

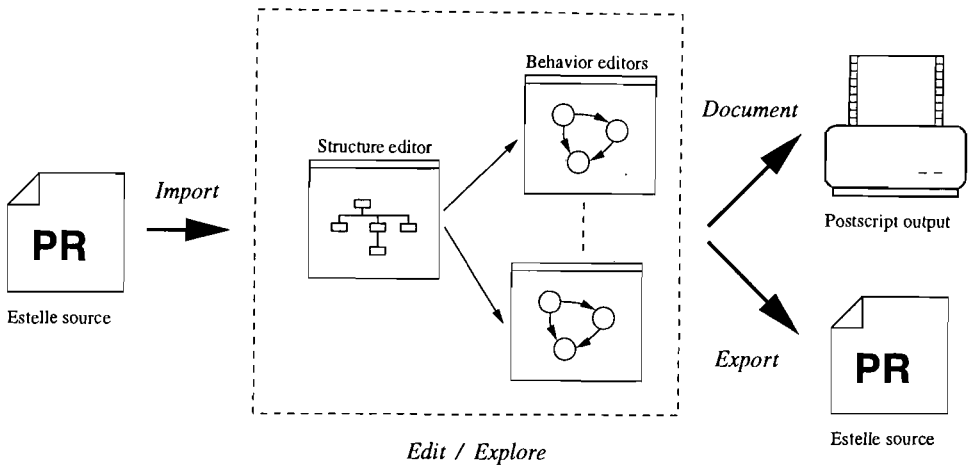


Figure 6.1: **The main functions of the Estelle editor.**

The *Import*, *Export* and *Document* functions are implemented as commands in the editor menu. *Edit* and *Explore* are two uses of the editor.

As an *editor* a new specification is created in the graphical form. When completed, it is *exported* to a file in Estelle/PR form for further processing by existing text-based Estelle tools.

As an *explorer* an existing Estelle/PR specification document is imported into the editor and the graphical form in order to help understanding and/or generate graphical documentation.

6.2 User's view

The Estelle editor is actually two editors:

- A **structure editor** in which an Estelle system structure can be defined, and
- A **behavior editor** in which the internal behavior of an Estelle module body is defined.

In addition, the behavior editor may generate a state-relation diagram from its defined behavior which is displayed in a **simplified automaton window**.

6.2.1 Structure editor

The structure editor defines the **structure tree diagram** of the Estelle/GR grammar. The **instance block diagram** is not currently supported.

The structure editor is also the "main" window, and is automatically opened when the editor is started. The general document-management functions of file handling and printing, as well as *import* and *export* of Estelle/PR specification documents, are invoked through its *command menu.* Behavior editors are also opened through its interface.

Figure 6.2 presents a screenshot of the editor interface. It consists of six components, which the behavior editor shares as well:

1. *Command menu* - A pull-down menu of commands.
2. *Tool bar* - Short-cut buttons for frequently-used menu commands.
3. *Symbol bar* - Buttons for the creation and editing of specification objects in the *workspace.*
4. *Workspace* - A scrollable, infinite canvas for defining a specification structure or module body behavior.
5. *Status window* - A window in which status messages, button explanations and the results of commands are displayed.
6. *File name / Module body name* - The identifying name of the specification or module body being edited in this window.

6.2.2 Behavior editor

The behavior editor defines the **transition part** of a module body in the Estelle/GR grammar using the transition tree notation.

A behavior editor may be opened for each module body declared in the structure tree diagram of the structure editor. Multiple behavior editors may be open simultaneously. Figure 6.3 presents a screenshot of the behavior editor.

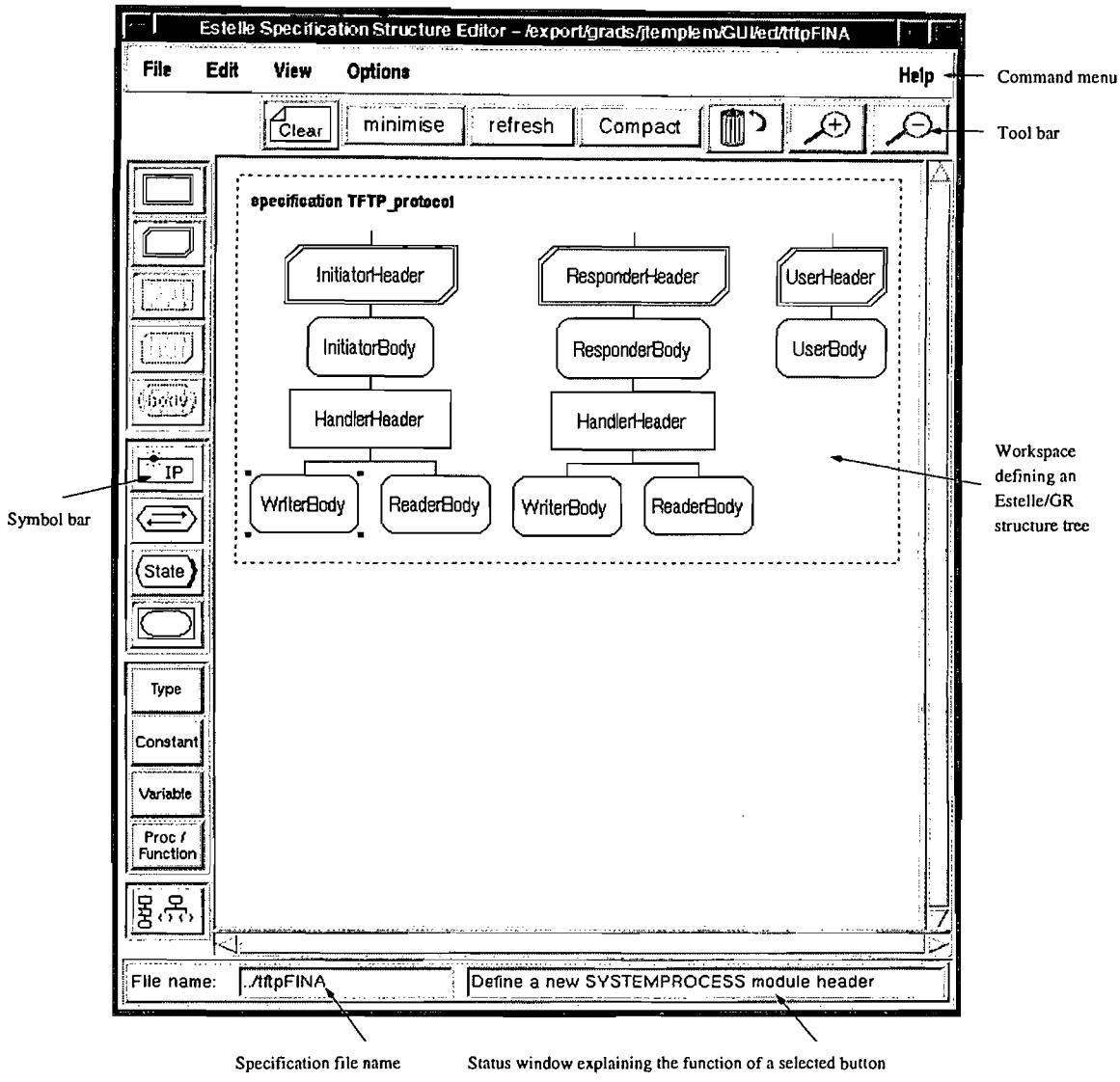


Figure 6.2: **The structure editor.** The structure editor defines the generic module hierarchy of an Estelle specification.

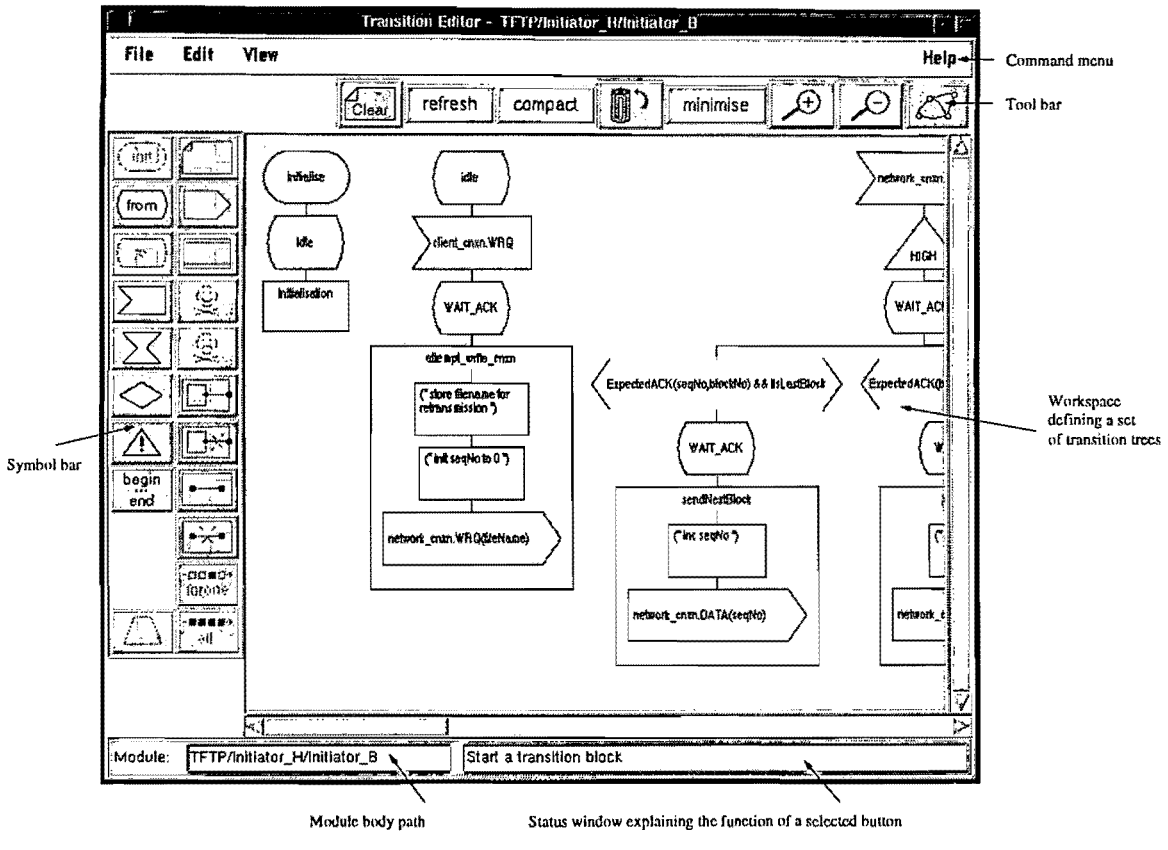


Figure 6.3: **The behavior editor.** The behavior editor defines the internal behavior of a module body.

6.2.3 Simplified automaton window

The set of transition trees in a behavior editor workspace uses a flow diagram notation to define an EFSM. The simplified automaton window of the editor returns to the conceptual model, and presents the EFSM as a simplified automaton which we call a *state-relation diagram.*

The simpler state-relation diagram provides an *overview* of module body behavior, and by way of dynamic links between the transitions of the automaton and editor windows, is also useful as an *index* of the detailed transition tree definitions. Figure shows a state-relation diagram in the simplified automaton window of the editor.

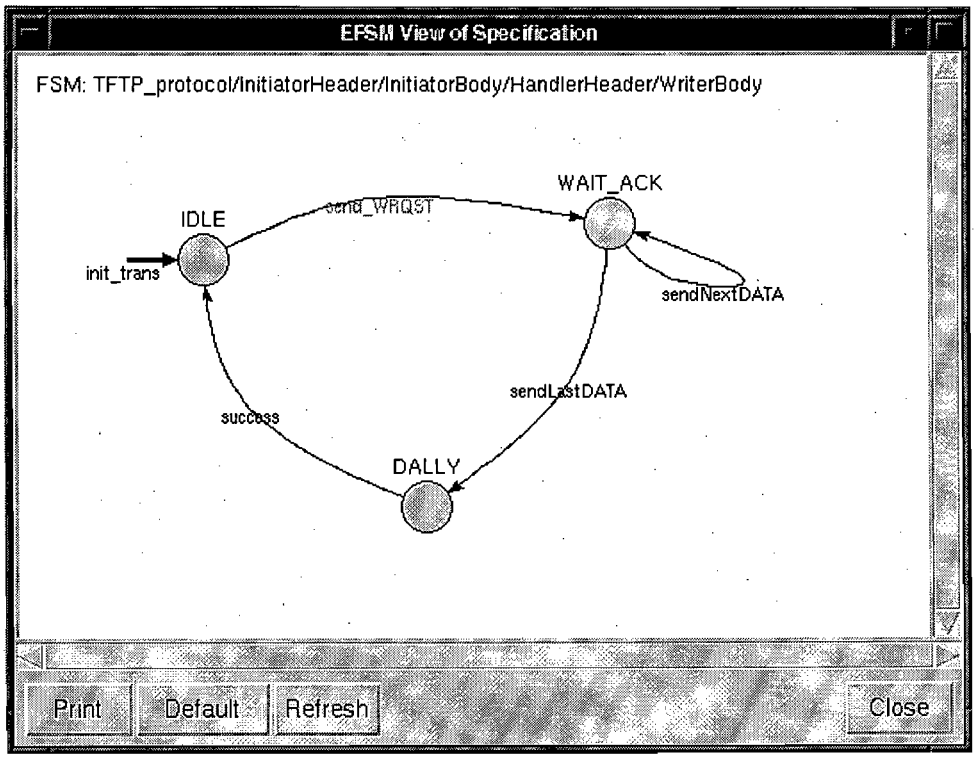


Figure 6.4: **The simplified automaton window.** The simplified automaton window displays a printable state-relation diagram that is dynamically linked to the behavior editor transition definitions.

6.2.3.1 State-relation diagram

A *state-relation diagram* is a simplified view of an automaton in which the number of transitions is reduced. Using a concept similar to the *macro-transition* defined in GROPE (see Section 4.1), all transitions with the same origin and destination states are grouped into a single *relation* between those states.

This results in a simplified view of an automaton, in which all the states are shown but there are at most two *relations* between the same origin-destination pair (in opposite directions). The full meaning of the non-simplified automaton is retained by labeling the relation with the names of all the grouped transitions.

6.2.3.2 Dynamic linking of the automaton window and behavior editor

The editor maintains dynamic links between the automaton and the transition tree definitions: Clicking on a transition name in the automaton centers the detailed transition tree definition in the behavior editor workspace. This turns the automaton view into an index of the complex transition tree definitions in the behavior editor workspace.

This linking also simplifies the exploration of a specification document, making it easier to assimilate the meaning of a module behavior. The simplified automaton provides an overview of the relationship of one transition to another, while at the same time the dynamic links to the behavior editor make it easy to recall the detailed transition definition.

The behavior editor and automaton window are also linked in the sense that they are always kept consistent, and changes to the behavior editor are immediately reflected in the state-relation diagram. This provides the protocol specifier with a good synopsis of the progress of the EFSM definition in the behavior editor.

6.2.3.3 Interface

The state-relation diagram of a module body is displayed in a separate window. The simplified automaton window is opened from the toolbar of the behavior editor.

States and transitions are initially placed in a geometric form, after which the user can place both the transition arcs and states as desired. These new positions are saved, and restored when the window is next opened.

The state-relation diagram displayed in the automaton window is printable.

6.3 Editor support for graphical design

The editor supports a *top-down* methodology of design where:

1. The generic module structure is defined in the structure editor;

followed by which

2. The internal behavior of each module body is defined in the behavior editor.

A structure or module body behavior is defined incrementally by the creation of graphical objects in the editor *workspace*. Each graphical object corresponds to an Estelle/GR graphical terminal symbol.

The editor is **syntax-directed** and prevents the use of incorrect graphical syntax when creating graphical objects. The attributes of each object objects are defined in a **structured editing interface** which is independent of a textual syntax and uses **dynamic scope calculation** to simplify object referencing.

These capabilities make the editor a vital part of effective specification design. They ensure that correct Estelle syntax and static semantics are used, which reduces both compilation and run-time errors and results in a more reliable specification.

6.3.1 Syntax-directed editing

The editor uses knowledge of the Estelle/GR syntax and static semantics to limit the actions and options available to the user, and thereby ensure the definition of a compileable specification.

The main method of doing this in the editor is by dynamically enabling and disabling the object creation buttons of the *symbol bar* depending on the object currently selected by the user. For example, the Estelle model prohibits the inclusion of both a **delay** and a **when** clause in a single transition definition. The behavior editor enforces this constraint by disabling the 'delay button' on the *symbol bar* if a **when** clause already exists in a selected

transition, and vice-versa. The structure editor imposes similar limitations on the creation of child modules to ensure the definition of a valid system structure.

The editor goes further, and attempts in places to anticipate the user's action. For example, a module header always has at least one associated module body. The editor pre-empts user creation of the first associated body by automatically creating it when a module header is created. This extends to encouraging good programming practices, by for example automatically adding a "provided otherwise" clause whenever a **provided** clause is created.

6.3.2 Structured object editing

The editor uses structured object editing to remind the user of the attributes required to define an object in the editor workspaces. This is opposed to "free-form" object editing used by tools such as Geode [Ver96], in which an object's attributes are specified in a textual syntax and then interpreted to create the graphical representation.

Structured editing has two main advantages in that it abstracts the syntax of textually-declared objects in the Estelle/GR, and simplifies object referencing.

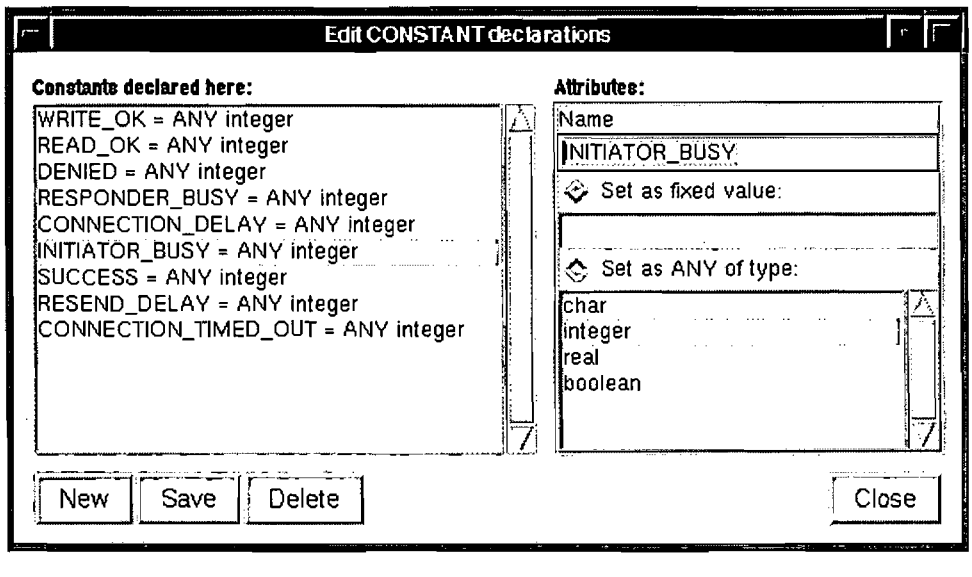


Figure 6.5: **Structured declaration of a CONSTANT.** The structure editing interface abstracts the textual syntax of Estelle/GR declarations.

A structured object definition eliminates the need for a defining syntax. This allows the user

to concentrate on the semantic meaning instead of trying to remember syntactic details. This is particularly important from the perspective that the Estelle/GR still contains a number of textual declarations, for example **types**, **variables** and **constants**. The structured editing environment provides an 'intermediate' form for these declarations, which although not graphical, does eliminate the details of the textual defining syntax.

The structured interface for the declaration of **constants** is shown in Figure 6.5. The left-hand pane lists declared constants and the right-hand pane contains the set of fields for defining the attributes of the constant selected on the left. The finite set of fields makes it easy for the user to remember what is required in a constant declaration, and does not require the use of a textual defining syntax.

6.3.3 Simplifying object referencing by dynamic scope calculation

Within the structured object interfaces the editor uses its knowledge of the Estelle scope rules to simplify object referencing by dynamically calculating identifier scopes.

The editor provides the user with selection lists of available identifiers instead of requiring the user to explicitly remember them. It further uses knowledge of the semantic rules of Estelle to filter the contents of these lists to statically valid choices.

When declaring variables such as in Figure 6.5, this scope calculation also prevents the declaration of duplicate identifiers.

For example, Figure 6.6 shows the construction of an **output** statement in the behavior editor. The left-hand list shows only interaction points in the current scope, and the user simply selects which to use in the statement. When an interaction point is selected, the editor uses knowledge of the relationship between interactions points, channels and roles to present a list of valid interactions sendable by that interaction point.

As the scope calculation also takes note of identifier occlusion when building identifier lists, it eliminates common 'out of scope' errors and mistaken references to accidentally re-defined identifiers which can result in run-time errors. The mouse-driven interface has the additional benefit of eliminating keystroke errors.

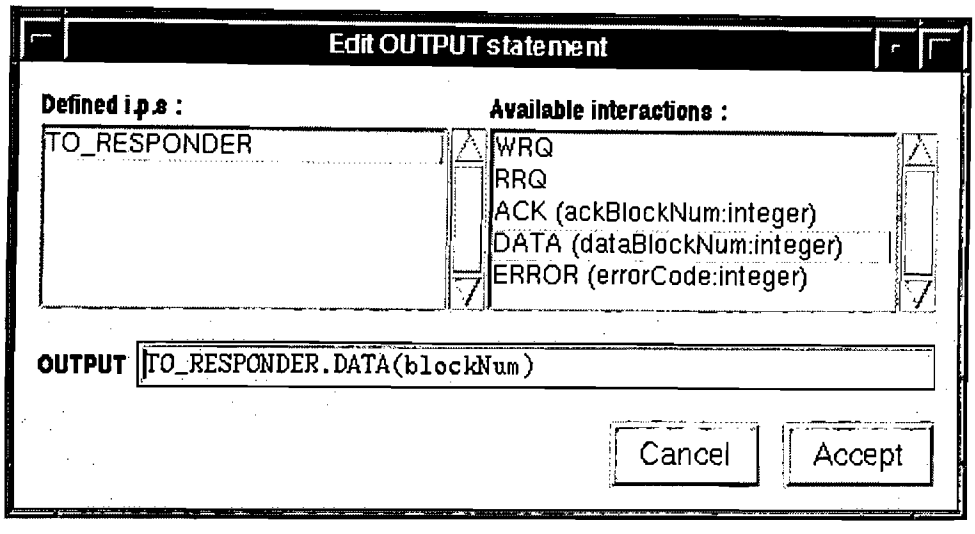


Figure 6.6: **Structured editing of an OUTPUT statement.** The structured editing interface only presents identifiers valid in the statement scope.

6.4 Editor features

In addition to the above features which support the design technique, the editor has a number of additional features which are designed to simplify editing and exploration of a specification document.

6.4.1 Multiple windows

The ability to open multiple behavior windows simultaneously allows related behaviors to be specified concurrently.

For example, the Initiator and Responder modules of the TFTP communicate directly with each other, and being able to view both behaviors on the screen simultaneously enables better understanding of how this communication takes place.

The permanently-open structure editor allows easy exploration of different module bodies, and serves as a reminder of how the module bodies of open behavior editors are related.

6.4.2 Global name replacement

The editor performs global name replacement when the name of an identifier is changed. All references to that identifier are automatically updated.

This is useful in cases where an interaction point or module variable is referenced many times in a single module body. In Estelle, *channel*, *type* and *constant* definitions are visible in all descendent modules of the defining module, thus there can be many references in many bodies to the same identifier. In this case, global name replacement is extremely useful.

6.4.3 Automatic document generation

The editor produces **postscript** output of the different diagrams as viewed in the structure editor, behavior editor and simplified automaton window. The user can select which parts of the specification to print. The system structure tree, individual module body transition definitions, and state-relation diagrams can all be printed individually.

The editor **does not** produce a complete Estelle/GR specification description. The diagrams are intended to be used as supporting documentation to the textual specification created by the *export* function.

This documentation is useful as it is generated automatically, which eliminates a previously time-consuming and tedious process. Importantly, the diagrams and the textual code will always be consistent, eliminating the ambiguity caused by human error and misrepresentation.

Once the editor is able to import an Estelle/PR document, it will be possible to use it to quickly and easily generate error-free graphical documentation for existing textual specifications.

6.5 Implementation language

The editor was implemented in **Tcl/Tk** version 8.0. **Tcl** is a UNIX-based scripting language with extensions (**Tk**) for graphical user interface (GUI) development.

Tcl/Tk was chosen for a number of reasons:

- It has rapid prototyping capabilities
- Programs are platform-independent as interpreters are supported for the native windowing environments of both UNIX and Windows.
- **Tk** is a well-developed graphical tool-kit dedicated to GUI development.
- It is freely available from Sun Microsystems.

Use of Tcl/Tk simplified development of the editor and means that it runs equally well under UNIX and Windows.

More information on **Tcl/Tk** can be found on the Internet at *http://www.sunscript.com.*

6.6 Ongoing work and future goals

Currently the editor is still in a prototype stage, and lacks a few significant functions.

Firstly, "cut, copy and paste" editing is severely limited and needs to be expanded. It will be possible to cut or copy parts of a module body definition between two behavior windows.

Secondly, it is not currently possible to import an existing Estelle/PR specification document. This is currently under development, and will enable the use of the editor interface to navigate and graphically document existing Estelle specifications.

Thirdly, the editor does not at the moment support the Estelle/GR **instance block diagram** notation. One of the future additions to the editor will be to add an environment for editing specifications using this notation.

Fourthly, the editor does not yet support the Estelle/GR commenting syntax.

When complete, the editor will be integrated into the XEdt Estelle tool-kit [INT97]. It will be possible to design and edit a specification in the graphical form, and then use the compilation, simulation and analysis tools of this toolkit to process the description further from within a single, integrated interface.

Chapter 7

Conclusion

In this dissertation we have defined an original graphical representation for the Estelle Formal Description Technique, and described a prototype editor based on this representation.

No formal, standard representation has to date been proposed for the Estelle FDT. Some informal representations have been explored, but all have been limited in the extent to which they could define a general ISO Estelle system.

Our new representation, monikered the **Estelle/GR**, builds on the ideas of these earlier attempts, as well as some of those used by other Formal Description Techniques, particularly the Specification and Description Language which is both widely used and similar to Estelle. The incorporation of this existing knowledge will aid in making the new graphical technique understandable to most FDT users.

The Estelle/GR takes advantage of the simplicity and expressiveness of graphical techniques to better fulfill the goals of the Formal Description Technique by creating concise and well-structured system definitions.

The main body of this dissertation rigorously defines a graphical syntax and semantics which is capable of expressing any Estelle system defined by ISO Estelle [ISO97]. This formal definition of the Estelle/GR will be submitted to the ISO for international standardization as a part of the Estelle FDT. We believe that standardization will enable the development of third party support tools for the graphical technique, which will stimulate a wider interest in, and application of, the Estelle FDT.

The prototype editor presented in Chapter 6 performs two essential functions of any editor for the Estelle/GR. Firstly it provides a modern graphical interface in which a protocol author may easily manipulate large graphical diagrams. Secondly it translates a graphical description into an equivalent textual description, which can be further processed by existing Estelle tools.

We were also able to use the editor interface to enhance the graphical technique. Some Estelle declarations have no graphical representation and are defined using the Estelle textual syntax. By way of structured object editing, the editor abstracts the syntactic details of these textual declarations, reducing this use of non-graphical constructs.

The Estelle/GR and prototype editor combine the many advantages of a graphical technique with the power of existing support tools for Estelle. This creates a powerful new graphical technique for the Estelle FDT, which we believe greatly increases its usefulness, and will stimulate its wider use in the future.

The Estelle/GR will be presented at the FORTE/PSTV conference on Formal Description Techniques to be held later this year, and it is already planned to include the prototype editor in the internationally-distributed XEdt toolkit [INT97] for Estelle.

Appendix A

The Trivial File Transfer Protocol

This appendix presents an illustrative implementation of the Trivial File Transfer Protocol (TFTP) using the Estelle/GR.

The protocol was specified using the Estelle prototype editor, and the graphical diagrams and textual Estelle listing generated by the editor are reproduced.

A.1 Introduction to the TFTP

The Trivial File Transfer Protocol [Chi94, Sol81] is a simple protocol for reading and writing files from/to a remote server.

Any transfer begins with the transmission of a request to read (READ_RQST) or write (WRITE_RQST) to the remote server. If the server grants the request, the connection is opened and the file is packetized and transmitted as sequentially numbered DATA packets. Each DATA packet must be acknowledged with an ACK packet before next DATA packet can be sent.

If a file is being written, the initiator of the connection sends DATA packets and receives ACK packets. If a file is being read, the responder sends DATA packets and receives ACK packets.

If either a DATA or an ACK packet is lost on the connection, the intended recipient will timeout and re-send his last packet, thus causing the sender of the lost packet to retransmit that packet.

Any other errors result in termination of the connection. An error is signalled by the sending of an ERROR packet, which is not acknowledged.

Successful completion of a transfer is recognised when the last DATA packet has been sent and the ACK for this packet has been received by the sender. The sender of the last ACK 'dallies' after sending it, in case the ACK is lost on the network and needs to be sent again. This is indicated by receipt of the last DATA packet again.

Connection termination is not acknowledged, therefore the lifetimes of the initiator and responder are limited by timeouts.

A.2 Our implementation

Our implementation of the TFTP specifies three asynchronous processes: a User, an Initiator and a Responder. The network medium is assumed to be error-free, thus the Initiator and Responder are connected directly between interaction points TO_INITIATOR and TO_RESPONDER. The User is an undefined systemprocess which sends READ_RQSTs and WRITE_RQSTs to the Initiator through interaction point CLIENT and grants or refuses permission to the Responder through interaction point SERVER.

Data transfer of a single is performed by a "Handler" which defines the Writer and Reader behaviors in separate bodies. The Initiator and Responder objects each have a single Handler which implements one end of the transfer logic.

On receipt of a request from the User, the Initiator (if it is not busy) creates a Handler with the appropriate body. The Handler has a single interaction point, which is attached to the environment, and the Initiator has no further role in the transfer. When the Handler has completed, it announces this to the Initiator by setting a shared flag, and the Initiator terminates the Handler and connection.

The Responder acts similarly. On receipt of a request from the Initiator Handler, it requests oermission from the Server. If this is granted, a handler is created, and the transfer continues between the two Handlers. The Handler is destroyed when it announces that it has finished.

The main advantage of specifying the TFTP with separate Reader and Writer behaviors is the isolation, and therefore clearer separation of these behaviors than if they were defined in

a single module. The behaviors are further simplfieid by the factoring out of the connection management behavior into the Initiator or Responder module.

Multiple connections are also easy to specify by adding multiple interaction points to the Initiator and Responder objects, and allowing the creation of a Handler for each available interaction point.

A.3 Graphical documentation

The following diagrams present the structure tree of the protocol, and simplified automaton for each active module body involved in the file writing behavior. In addition, the detailed transition tree definition of the writer behavior in the Initiator is produced as an example of a complete module body transition part definition, for comparison with the equivalent simplified automaton.

Note that these diagrams are produced with the *document* function of the editor, therefore they are not complete definitions. The full specification is listed textually in the following section.

Figure A.1 presents the editor structure diagram. A complete definition can also be seen in Figure 5.1 on Page 51, as well as in its initial state in Figure 5.2.

Figure A.2 presents the automaton for the Initiator module. This contains the initiator connection management.

Figure A.3 presents the automaton for the Responder module. This contains the responder connection management and interaction with the Server object.

Figure A.4 presents the automaton for the Writer body of the Handler of the Initiator.

Figure A.5 presents the automaton for the Writer body of the Handler of the Responder.

Figure A.6 presents the full of transition trees defining the Writer body of the Handler of the Responder.

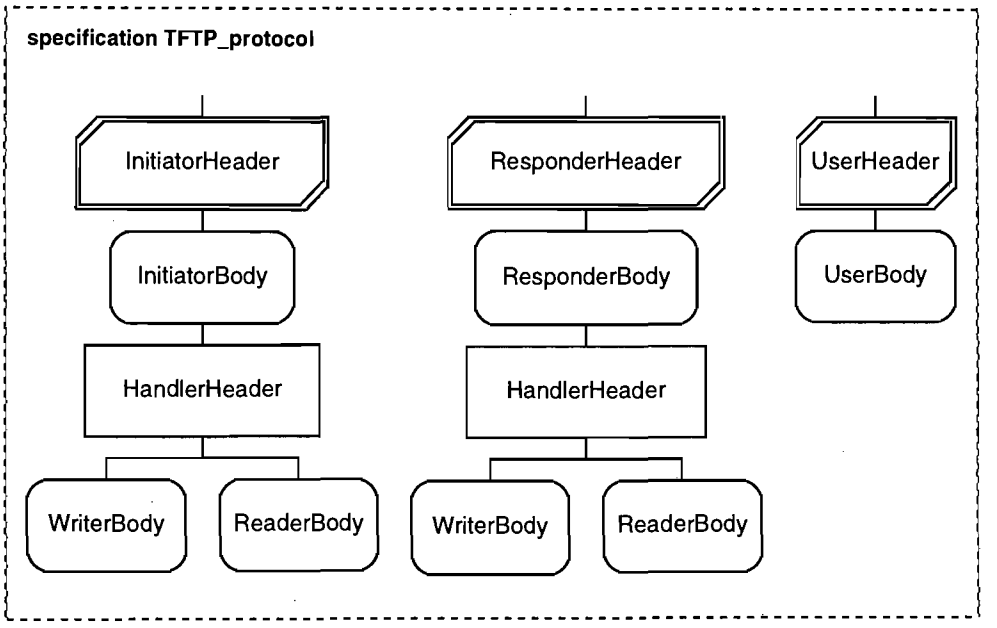


Figure A.1: TFTP Structure tree

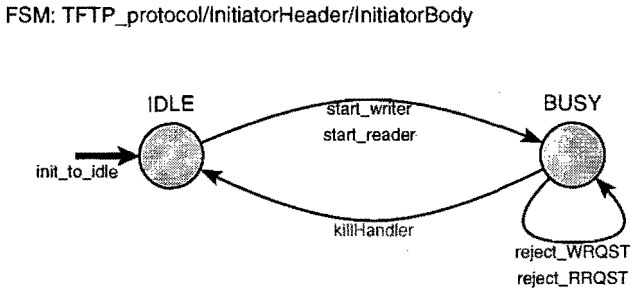


Figure A.2: TFTP Initiator connection management

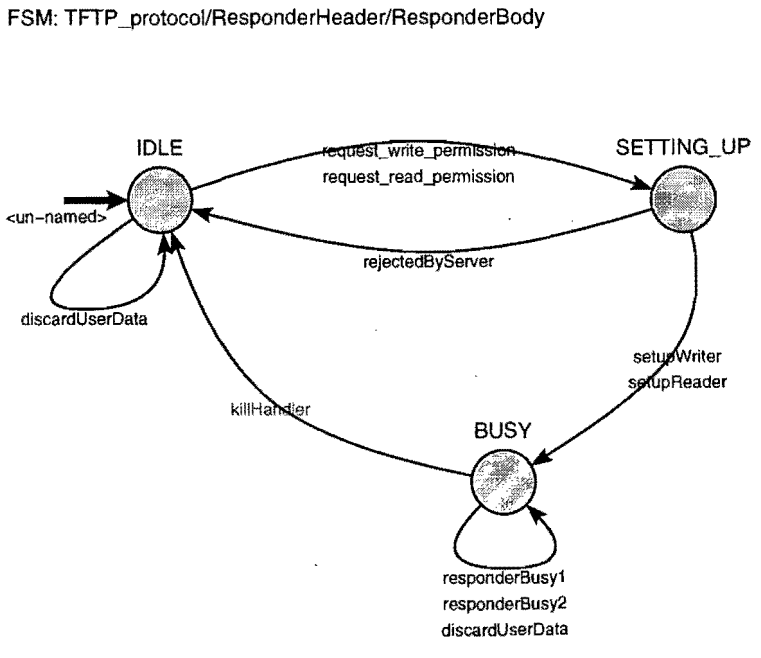


Figure A.3: TFTP Responder connection management

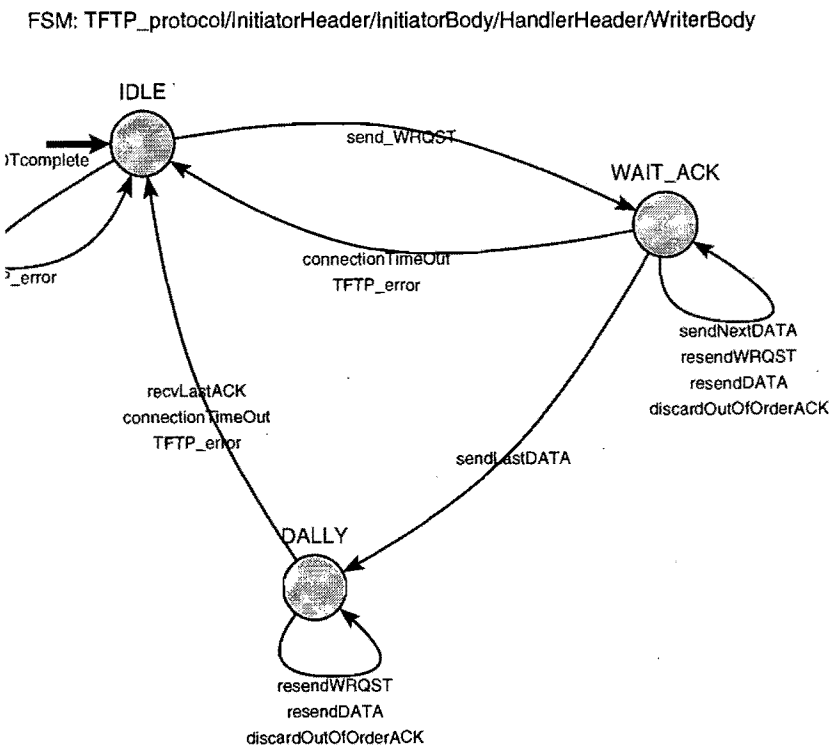


Figure A.4: TFTP Initiator writer behavior

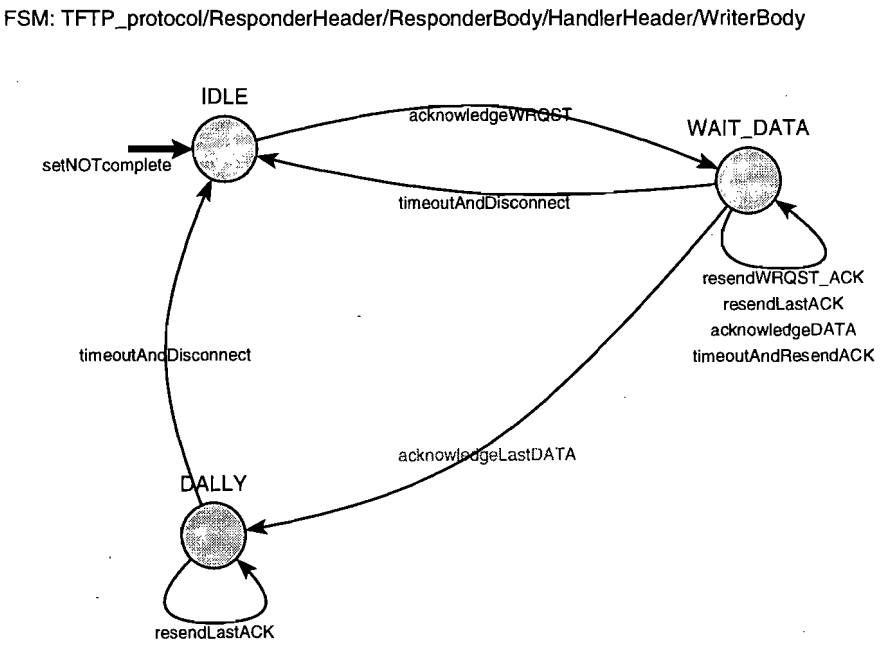


Figure A.5: TFTP Responder writer behavior

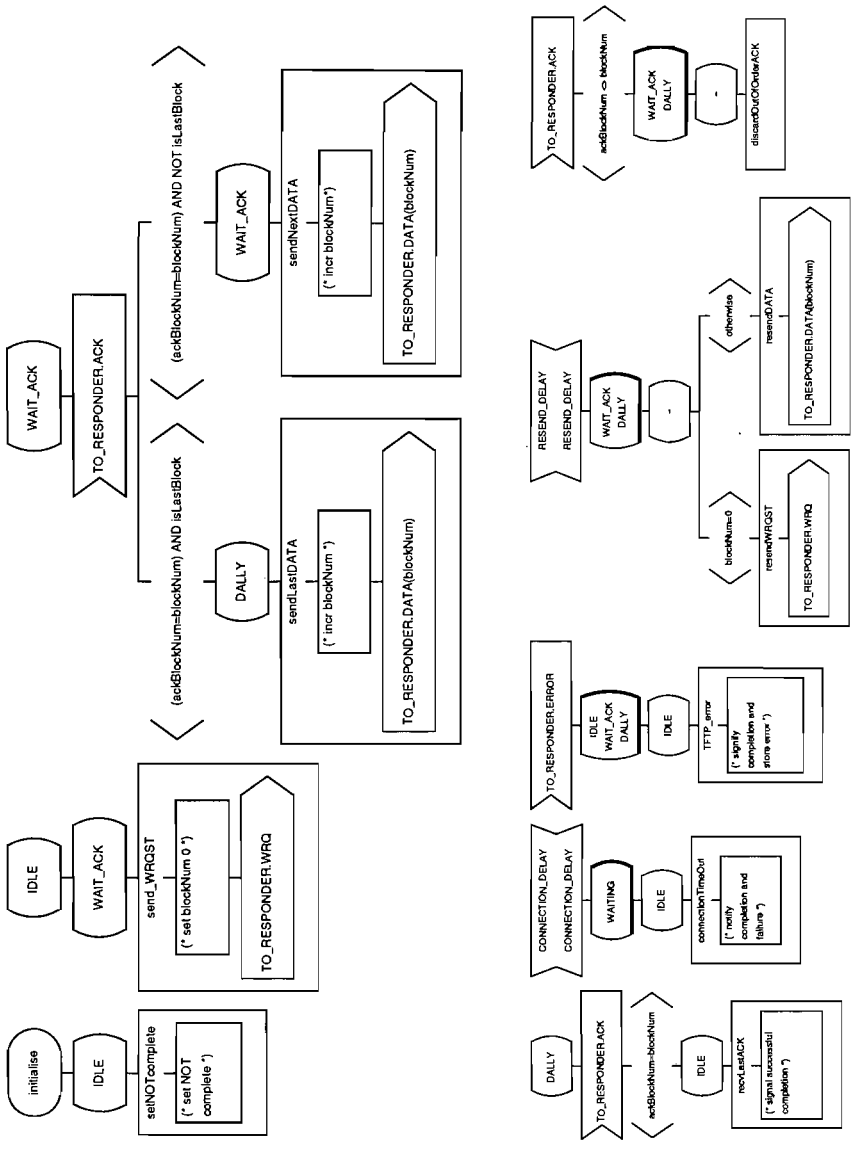


Figure A.6: Initiator writer behavior - transition trees

A.4 Textual Estelle listing

This code was generated in the editor from the same Estelle/GR document as the diagrams above, and is consistent with these representations.

```
(*
#########################################################
  ESTELLE SOURCE CODE
  FILE     : tftp.stl
  GENERATED: Tue Jul 28 18:08:24 SAT 1998
#########################################################
*)


SPECIFICATION TFTP_protocol ;
default individual queue ;
timescale seconds ;

{ CONSTANT DECLARATION PART TFTP_protocol }
   const
      WRITE_OK = any integer;
      READ_OK = any integer;
      DENIED = any integer;
      RESPONDER_BUSY = any integer;
      CONNECTION_DELAY = any integer;
      INITIATOR_BUSY = any integer;
      SUCCESS = any integer;
      RESEND_DELAY = any integer;
      CONNECTION_TIMED_OUT = any integer;

{ TYPE DECLARATION PART TFTP_protocol }

{ VARIABLE DECLARATION PART TFTP_protocol }

{ PROCEDURE&FUNCTION DECLARATION PART TFTP_protocol  }
```

```
{ STATE DECLARATION PART TFTP_protocol }

{ CHANNEL DECLARATION PART TFTP_protocol }

   channel user_access_point (user,provider);
      by user:
         READ_RQST ;
         WRITE_RQST ;
      by provider:
         RESPONSE (returnCode:integer);

   channel network_access_point (initiator,responder);
      by initiator:
         WRQ ;
         RRQ ;
      by initiator,responder:
         ACK (ackBlockNum:integer);
         DATA (dataBlockNum:integer);
         ERROR (errorCode:integer);

{ INTERACTION POINT DECLARATION PART TFTP_protocol }

{ MODULE HEADER DECLARATION PART TFTP_protocol }
   module InitiatorHeader systemprocess ;
      ip
         U : user_access_point (provider)  ;
         N : network_access_point (initiator)  ;
   end;  { MODULE HEADER InitiatorHeader }
   module ResponderHeader systemprocess ;
      ip
         N : network_access_point (responder)  ;
         U : user_access_point (user)  ;
   end;  { MODULE HEADER ResponderHeader }
```

```
   module UserHeader systemprocess ;
      ip
         SERVER : user_access_point (provider)  ;
         CLIENT : user_access_point (user)  ;
   end;  { MODULE HEADER UserHeader }

{ MODULE BODY DECLARATION PART TFTP_protocol }
   body InitiatorBody for InitiatorHeader;

   { CONSTANT DECLARATION PART InitiatorBody }

   { TYPE DECLARATION PART InitiatorBody }

   { VARIABLE DECLARATION PART InitiatorBody }
      var
         userResult : integer;

   { PROCEDURE&FUNCTION DECLARATION PART InitiatorBody  }

   { STATE DECLARATION PART InitiatorBody }
      state IDLE,BUSY;

   { CHANNEL DECLARATION PART InitiatorBody }

   { INTERACTION POINT DECLARATION PART InitiatorBody }

   { MODULE HEADER DECLARATION PART InitiatorBody }
      module HandlerHeader activity ;
         ip
            TO_RESPONDER : network_access_point (initiator)  ;
         export
            complete : boolean;
            result : integer;
```

```
    end;  { MODULE HEADER HandlerHeader }

{ MODULE BODY DECLARATION PART InitiatorBody }
    body WriterBody for HandlerHeader;

    { CONSTANT DECLARATION PART WriterBody }

    { TYPE DECLARATION PART WriterBody }

    { VARIABLE DECLARATION PART WriterBody }
        var
            blockNum : integer;

    { PROCEDURE&FUNCTION DECLARATION PART WriterBody  }
        pure function isLastBlock  : boolean; external;

    { STATE DECLARATION PART WriterBody }
        state IDLE,WAIT_ACK,DALLY;
        stateset
            WAITING = [WAIT_ACK, DALLY];

    { CHANNEL DECLARATION PART WriterBody }

    { INTERACTION POINT DECLARATION PART WriterBody }

    { MODULE HEADER DECLARATION PART WriterBody }

    { MODULE BODY DECLARATION PART WriterBody }

    { MODULE VARIABLE DECLARATION PART WriterBody }

    { TRANSITION DECLARATION PART }
    initialize
```

```
      TO IDLE
         name setNOTcomplete : begin
            (* set NOT complete *)

            complete := false;
         end;
   trans
      FROM IDLE
         TO WAIT_ACK
            name send_WRQST : begin
               (* set blockNum 0 *)

               blockNum := 0;
               OUTPUT TO_RESPONDER.WRQ ;
            end;
   trans
      FROM WAIT_ACK
         WHEN TO_RESPONDER.ACK
            PROVIDED (ackBlockNum=blockNum) AND isLastBlock
               TO DALLY
                  name sendLastDATA : begin
                     (* incr blockNum *)

                     blockNum := blockNum+1;
                     OUTPUT TO_RESPONDER.DATA(blockNum) ;
                  end;
            PROVIDED (ackBlockNum=blockNum) AND NOT isLastBlock
               TO WAIT_ACK
                  name sendNextDATA : begin
                     (* incr blockNum*)

                     blockNum := blockNum + 1;
                     OUTPUT TO_RESPONDER.DATA(blockNum) ;
```

```
                end;
        begin end;
trans
   FROM DALLY
      WHEN TO_RESPONDER.ACK
         PROVIDED ackBlockNum=blockNum
            TO IDLE
               name recvLastACK : begin
                  (* signal successful completion *)

                  complete := true;
                  result := SUCCESS;
               end;
trans
   DELAY (CONNECTION_DELAY, CONNECTION_DELAY )
      FROM WAITING
         TO IDLE
            name connectionTimeOut : begin
               (* notify completion
                  and failure *)

               complete := true;
               result := CONNECTION_TIMED_OUT;
            end;
trans
   WHEN TO_RESPONDER.ERROR
      FROM IDLE, WAIT_ACK, DALLY
         TO IDLE
            name TFTP_error : begin
               (* signify completion and
                  store error *)

               complete := true;
```

```
                result := errorCode;
             end;
trans
   DELAY (RESEND_DELAY, RESEND_DELAY )
      FROM WAIT_ACK, DALLY
         TO same
            PROVIDED blockNum=0
               name resendWRQST : begin
                  OUTPUT TO_RESPONDER.WRQ ;
               end;
            PROVIDED otherwise
               name resendDATA : begin
                  OUTPUT TO_RESPONDER.DATA(blockNum) ;
               end;
      begin end;
trans
   WHEN TO_RESPONDER.ACK
      PROVIDED ackBlockNum <> blockNum
         FROM WAIT_ACK, DALLY
            TO same
               name discardOutOfOrderACK : begin
               end;
end;  { MODULE BODY WriterBody }
body ReaderBody for HandlerHeader;

{ CONSTANT DECLARATION PART ReaderBody }

{ TYPE DECLARATION PART ReaderBody }

{ VARIABLE DECLARATION PART ReaderBody }
   var
      blockNum : integer;
```

```
{ PROCEDURE&FUNCTION DECLARATION PART ReaderBody  }
   pure function isLastBlock  : boolean; external;

{ STATE DECLARATION PART ReaderBody }
   state IDLE,WAIT_DATA,DALLY;

{ CHANNEL DECLARATION PART ReaderBody }

{ INTERACTION POINT DECLARATION PART ReaderBody }

{ MODULE HEADER DECLARATION PART ReaderBody }

{ MODULE BODY DECLARATION PART ReaderBody }

{ MODULE VARIABLE DECLARATION PART ReaderBody }

{ TRANSITION DECLARATION PART }
initialize
   TO IDLE
      name init_trans : begin
         (* set complete false *)

         complete := true;
      end;
trans
   FROM IDLE
      TO WAIT_DATA
         name sendRRQ : begin
            (* set blockNum 0 *)

            blockNum := 0;
            OUTPUT TO_RESPONDER.RRQ ;
         end;
```

```
trans
   DELAY (RESEND_DELAY, RESEND_DELAY ) ·
      FROM WAIT_DATA
         TO same
            PROVIDED blockNum=0
               name resendRRQ : begin
                  OUTPUT TO_RESPONDER.RRQ ;
               end;
            PROVIDED otherwise
               name resendACK : begin
                  OUTPUT TO_RESPONDER.ACK(blockNum) ;
               end;
      begin end;
trans
   DELAY (CONNECTION_DELAY, CONNECTION_DELAY )
      FROM WAIT_DATA, DALLY
         TO IDLE
            name connectionTimeout : begin
               (* set complete true*)

               complete := true;
            end;
trans
   WHEN TO_RESPONDER.ERROR
      FROM IDLE, WAIT_DATA, DALLY
         TO IDLE
            name handleError : begin
               (* set complete true *)

               complete := true;
               (* return error *)

               result := errorCode;
```

```
            end;
trans
   FROM WAIT_DATA
      WHEN TO_RESPONDER.DATA
         PROVIDED isLastBlock AND (dataBlockNum=blockNum+1)
            TO DALLY
               name sendLastACK : begin
                  (* incr blockNum *)

                  blockNum := blockNum + 1;
                  OUTPUT TO_RESPONDER.ACK(blockNum) ;
               end;
         PROVIDED NOT isLastBlock AND (dataBlockNum=blockNum+1)
            TO WAIT_DATA
               name sendACK : begin
                  (* incr blockNum *)
                  OUTPUT TO_RESPONDER.ACK(blockNum) ;
               end;
         PROVIDED otherwise
            name discardBadDATA : begin
            end;
      begin end;
trans
   FROM DALLY
      TO DALLY
         WHEN TO_RESPONDER.DATA
            PROVIDED blockNum=dataBlockNum
               name resendLastACK : begin
               end;
            PROVIDED otherwise
               name discardBadDATA2 : begin
               end;
      begin end;
```

```
   end;  { MODULE BODY ReaderBody }

{ MODULE VARIABLE DECLARATION PART InitiatorBody }
   modvar
      handler : HandlerHeader;

{ TRANSITION DECLARATION PART }
initialize
   TO IDLE
      name init_to_idle : begin
      end;
trans
   WHEN U.WRITE_RQST
      FROM IDLE
         TO BUSY
            name start_writer : begin
               INIT handler WITH WriterBody ;
               ATTACH N TO handler.TO_RESPONDER ;
            end;
      FROM BUSY
         TO same
            name reject_WRQST : begin
               OUTPUT U.RESPONSE(INITIATOR_BUSY) ;
            end;
      begin end;
trans
   WHEN U.READ_RQST
      FROM IDLE
         TO BUSY
            name start_reader : begin
               INIT handler WITH ReaderBody ;
               ATTACH N TO handler.TO_RESPONDER ;
            end;
```

```
      FROM BUSY
         TO same
            name reject_RRQST : begin
               OUTPUT U.RESPONSE(INITIATOR_BUSY) ;
            end;
      begin end;
trans
   PROVIDED handler.complete
      FROM BUSY
         TO IDLE
            name killHandler : begin
               TERMINATE handler  ;
               OUTPUT U.RESPONSE(handler.result) ;
            end;
end;  { MODULE BODY InitiatorBody }
body ResponderBody for ResponderHeader;

{ CONSTANT DECLARATION PART ResponderBody }

{ TYPE DECLARATION PART ResponderBody }

{ VARIABLE DECLARATION PART ResponderBody }
   var
      oldBusy : boolean;

{ PROCEDURE&FUNCTION DECLARATION PART ResponderBody  }

{ STATE DECLARATION PART ResponderBody }
   state IDLE,BUSY,SETTING_UP;
   stateset
      NOT_IDLE = [BUSY];

{ CHANNEL DECLARATION PART ResponderBody }
```

```
{ INTERACTION POINT DECLARATION PART ResponderBody }

{ MODULE HEADER DECLARATION PART ResponderBody }
   module HandlerHeader activity ;
      ip
         TO_INITIATOR : network_access_point (responder)  ;
      export
         complete : boolean;
   end;  { MODULE HEADER HandlerHeader }

{ MODULE BODY DECLARATION PART ResponderBody }
   body WriterBody for HandlerHeader;

   { CONSTANT DECLARATION PART WriterBody }

   { TYPE DECLARATION PART WriterBody }

   { VARIABLE DECLARATION PART WriterBody }
      var
         blockNum : integer;

   { PROCEDURE&FUNCTION DECLARATION PART WriterBody  }
      pure function isLastBlock  : boolean; external;

   { STATE DECLARATION PART WriterBody }
      state IDLE,WAIT_DATA,DALLY;

   { CHANNEL DECLARATION PART WriterBody }

   { INTERACTION POINT DECLARATION PART WriterBody }

   { MODULE HEADER DECLARATION PART WriterBody }
```

```
{ MODULE BODY DECLARATION PART WriterBody }

{ MODULE VARIABLE DECLARATION PART WriterBody }

{ TRANSITION DECLARATION PART }
initialize
   TO IDLE
      name setNOTcomplete : begin
         (* set as incomplete *)

         complete := false;
      end;
trans
   FROM IDLE
      TO WAIT_DATA
         name acknowledgeWRQST : begin
            OUTPUT TO_INITIATOR.ACK(0) ;
            (* set blockNum 0*)

            blockNum := 0;
         end;
trans
   FROM WAIT_DATA
      TO WAIT_DATA
         name resendWRQST_ACK : begin
            OUTPUT TO_INITIATOR.ACK(0) ;
         end;
trans
   FROM WAIT_DATA, DALLY
      TO same
         WHEN TO_INITIATOR.DATA
            PROVIDED dataBlockNum=blockNum
```

```
                name resendLastACK : begin
                   OUTPUT TO_INITIATOR.ACK(blockNum) ;
                end;
trans
   FROM WAIT_DATA
      WHEN TO_INITIATOR.DATA
         PROVIDED (dataBlockNum=blockNum+1) AND isLastBlock
            TO DALLY
               name acknowledgeLastDATA : begin
                  (* inc blockNum *)

                  blockNum := blockNum+1;
                  OUTPUT TO_INITIATOR.ACK(blockNum) ;
               end;
         PROVIDED (dataBlockNum=blockNum+1) AND NOT isLastBlock
            TO WAIT_DATA
               name acknowledgeDATA : begin
                  (* incr blockNum *)

                  blockNum :=blockNum+1;
                  OUTPUT TO_INITIATOR.ACK(blockNum) ;
               end;
      begin end;
trans
   DELAY (CONNECTION_DELAY, CONNECTION_DELAY )
      FROM WAIT_DATA, DALLY
         TO IDLE
            name timeoutAndDisconnect : begin
               (* set complete true*)

               complete := true;
            end;
trans
```

```
    DELAY (RESEND_DELAY, RESEND_DELAY )
       FROM WAIT_DATA
          TO same
             name timeoutAndResendACK : begin
                OUTPUT TO_INITIATOR.ACK(blockNum) ;
             end;
end;  { MODULE BODY WriterBody }
body ReaderBody for HandlerHeader;

{ CONSTANT DECLARATION PART ReaderBody }

{ TYPE DECLARATION PART ReaderBody }

{ VARIABLE DECLARATION PART ReaderBody }
   var
      blockNum : integer;

{ PROCEDURE&FUNCTION DECLARATION PART ReaderBody  }
   pure function isLastBlock  : boolean; external;

{ STATE DECLARATION PART ReaderBody }
   state IDLE,WAIT_ACK,DALLY;

{ CHANNEL DECLARATION PART ReaderBody }

{ INTERACTION POINT DECLARATION PART ReaderBody }

{ MODULE HEADER DECLARATION PART ReaderBody }

{ MODULE BODY DECLARATION PART ReaderBody }

{ MODULE VARIABLE DECLARATION PART ReaderBody }
```

```
{ TRANSITION DECLARATION PART }
initialize
   TO IDLE
      name setNOTcomplete : begin
         (* set not complete *)

         complete := false;
      end;
trans
   WHEN TO_INITIATOR.RRQ
      FROM IDLE
         TO WAIT_ACK
            name sendFirstDATA : begin
               (* set blockNum 1 *)

               blockNum := 1;
               OUTPUT TO_INITIATOR.DATA(blockNum) ;
            end;
      FROM WAIT_ACK, DALLY
         TO same
            name discardBadRRQ : begin
            end;
      begin end;
trans
   DELAY (RESEND_DELAY, RESEND_DELAY )
      FROM WAIT_ACK, DALLY
         TO same
            name resendDATA : begin
               OUTPUT TO_INITIATOR.DATA(blockNum) ;
            end;
trans
   DELAY (CONNECTION_DELAY, CONNECTION_DELAY )
      FROM WAIT_ACK, DALLY
```

```
            TO IDLE
               name connectionTimeOut : begin
                  (* set complete true *)

                  complete := true;
               end;
      trans
         FROM WAIT_ACK
            WHEN TO_INITIATOR.ACK
               PROVIDED (ackBlockNum=blockNum) AND isLastBlock
                  TO DALLY
                     name sendLastDATA : begin
                        (* incr blockNum *)

                        blockNum := blockNum+1;
                        OUTPUT TO_INITIATOR.DATA(blockNum) ;
                     end;
               PROVIDED (blockNum=ackBlockNum) AND NOT isLastBlock
                  TO WAIT_ACK
                     name sendDATA : begin
                        (* incr blockNum *)

                        blockNum := blockNum + 1;
                        OUTPUT TO_INITIATOR.ACK(blockNum) ;
                     end;
               PROVIDED otherwise
                  name discardBadACK : begin
                  end;
            begin end;
      trans
         FROM DALLY
            WHEN TO_INITIATOR.ACK
               PROVIDED ackBlockNum=blockNum
```

```
                TO IDLE
                   name setComplete : begin
                      (* *)

                      complete := true;
                   end;
             PROVIDED otherwise
                TO same
                   name discardBadACK2 : begin
                   end;
          begin end;
   end;  { MODULE BODY ReaderBody }

{ MODULE VARIABLE DECLARATION PART ResponderBody }
   modvar
      handler : HandlerHeader;

{ TRANSITION DECLARATION PART }
initialize
   TO IDLE
      begin end;
trans
   WHEN N.WRQ
      FROM IDLE
         TO SETTING_UP
            name request_write_permission : begin
               OUTPUT U.WRITE_RQST ;
            end;
      FROM NOT_IDLE
         TO same
            name responderBusy1 : begin
               OUTPUT N.ERROR(RESPONDER_BUSY) ;
            end;
```

```
        begin end;
  trans
     WHEN N.RRQ
        FROM IDLE
           TO SETTING_UP
              name request_read_permission : begin
                 OUTPUT U.READ_RQST ;
              end;
        FROM NOT_IDLE
           TO same
              name responderBusy2 : begin
                 OUTPUT N.ERROR(RESPONDER_BUSY) ;
              end;
        begin end;
  trans
     WHEN U.RESPONSE
        FROM SETTING_UP
           PROVIDED returnCode = WRITE_OK
              TO BUSY
                 name setupWriter : begin
                    INIT handler WITH WriterBody ;
                    ATTACH N TO handler.TO_INITIATOR ;
                 end;
           PROVIDED returnCode = READ_OK
              TO BUSY
                 name setupReader : begin
                 end;
           PROVIDED otherwise
              TO IDLE
                 name rejectedByServer : begin
                    OUTPUT N.ERROR(DENIED) ;
                 end;
           begin end;
```

```
        FROM IDLE, BUSY
           TO same
              name discardUserData : begin
              end;
        begin end;
   trans
      PROVIDED handler.complete
        FROM BUSY
           TO IDLE
              name killHandler : begin
                 TERMINATE handler  ;
              end;
   end;  { MODULE BODY ResponderBody }
   body UserBody for UserHeader;
   external;

{ MODULE VARIABLE DECLARATION PART TFTP_protocol }
   modvar
      Initiator : InitiatorHeader;
      Responder : ResponderHeader;

{ TRANSITION DECLARATION PART }
initialize
   name trans1 : begin
      INIT Initiator WITH InitiatorBody ;
      INIT Responder WITH ResponderBody ;
      CONNECT Initiator.N TO Responder.N ;
   end;
end. { end of specification TFTP_protocol }
```

Appendix B

Collected syntax

B.1 Comments

GR-comment =
    GR-comment-symbol
    CONTAINS
        GR-comment-text
    CONNECTED TO
        [GR-commentable-symbol] .

GR-comment-symbol =  .

GR-comment-text = < *text* > .

GR-commentable-symbol =
    GR-external-body-symbol
    | GR-condition-symbol
    | GR-from-state-symbol
    | GR-from-state-set-symbol
    | GR-to-state-symbol
    | GR-any-symbol
    | GR-delay-symbol

| GR-when-symbol
| GR-priority-symbol
| GR-procedure-reference-symbol
| GR-output-symbol
| GR-text-symbol
| GR-attach-symbol
| GR-detach-symbol
| GR-connect-symbol
| GR-disconnect-symbol
| GR-initialize-symbol
| GR-release-symbol
| GR-terminate-symbol .

B.2 Diagram partitioning

GR-frame-symbol =  .

B.3 Structure of an Estelle/GR specification diagram

GR-specification-diagram =
    GR-structure-definition-part
    { GR-body-definition-part } .

GR-structure-definition-part =
    GR-frame-symbol
    CONTAINS (
        GR-structure-definition
        [GR-specification-comment]) .

GR-structure-definition =
    GR-structure-tree-diagram
    | GR-instance-block-diagram .

GR-specification-comment =
    GR-comment .

B.4 Structure tree diagram

GR-structure-tree-diagram =
    GR-specification-module
    IS FOLLOWED BY
        { GR-child-module } .

GR-specification-module =
    GR-specification-symbol
    CONTAINS
        PR-specification-identifier
    IS ASSOCIATED WITH (
        [PR-default-options]
        [PR-time-options]) .

GR-specification-symbol =
    GR-systemactivity-symbol
    | GR-systemprocess-symbol
    | GR-unattributed-symbol .

GR-systemactivity-symbol =  .

GR-systemprocess-symbol =  .

GR-unattributed-symbol =  .

GR-child-module =
    GR-module-header
    IS FOLLOWED BY

+{ GR-module-body } .

B.4.1 Module header definition

GR-module-header =
    GR-header-symbol
    CONTAINS
        PR-header-identifier [PR-parameter-list]
    IS ASSOCIATED WITH
        { (GR-individual-queue-symbol
        | GR-common-queue-symbol
        | GR-default-queue-symbol)
        GR-ip-identifier [GR-ip-dimensions] ":"
        GR-channel-identifier "<" GR-role-identifier ">" }
    IS ASSOCIATED WITH
        ["export" + PR-exported-variable-declaration] .

GR-header-symbol =
    GR-systemprocess-symbol
    | GR-systemactivity-symbol
    | GR-process-symbol
    | GR-activity-symbol .

GR-process-symbol =  .

GR-activity-symbol =  .

GR-common-queue-symbol =  .

GR-individual-queue-symbol =  .

GR-default-queue-symbol =  .

GR-ip-dimensions = "[" PR-index-type-list "]" .

GR-channel-identfier = PR-identifier .

GR-role-identfier = PR-identifier .

B.4.2 Module body declaration

GR-module-body =
(GR-body-declaration
IS FOLLOWED BY
{ GR-child-module })
| GR-external-body .

GR-body-declaration =
GR-body-symbol
CONTAINS
PR-body-identifier
IS ASSOCIATED WITH
PR-interaction-point-declaration-part

GR-external-body =
GR-external-body-symbol
CONTAINS
PR-body-identifier .

GR-body-symbol =  .

GR-external-body-symbol =  .

B.5 Instance block diagram

GR-instance-block-diagram = GR-specification-instance .

GR-specification-instance =
    GR-specification-symbol
    IS ASSOCIATED WITH
        PR-specification-identifier
        [PR-default-options]
        [PR-time-options])
    CONTAINS (
        [GR-specification-comment]
        GR-instance-substructure) .

GR-specification-symbol =
    GR-systemactivity-symbol
    | GR-systemprocess-symbol
    | GR-unattributed-symbol .

GR-specification-comment =
    GR-comment .

B.5.1 Instance substructure

GR-instance-substructure =
    { GR-child-instance }
    [GR-channel-definition-part]
    { GR-attach-segment }
    { GR-connect-segment }
    { GR-ip-declaration } .

GR-channel-definition-part =
    GR-declaration-symbol

CONTAINS

PR-channel-definition .

GR-declaration-symbol =  .

The following Estelle/PR syntax channel definition is copied directly from [ISO97] § 7.3.4.1 for reference purposes.

PR-channel-definition = PR-channel-heading PR-channel-block .

PR-channel-heading = "channel" PR-channel-identifier "(" PR-role-list ")" ";" .

PR-channel-identifier = PR-identifier .

PR-role-list = PR-identifier "," PR-identifier .

B.5.2 Interaction point declaration

GR-ip-declaration =

GR-ip-symbol

IS ASSOCIATED WITH

PR-identifier [GR-ip-dimensions]

"<" GR-role-identifier ">" .

GR-ip-symbol =

GR-common-queue-symbol

| GR-individual-queue-symbol

| GR-default-queue-symbol .

B.5.3 Child instance

GR-child-instance =

GR-header-symbol
IS ASSOCIATED WITH (
GR-instance-header-part
{ GR-ip-declaration })
CONTAINS
[GR-instance-comment]
(GR-instance-substructure | "external") .

GR-instance-header-part =
GR-instance-identifier [GR-instance-dimensions]
[GR-instance-exported-variables]

GR-instance-identifier = PR-identifier .

GR-instance-dimensions = "[" PR-index-type-list "]" .

GR-instance-parameters = "(" PR-parameter-list ")" .

GR-instance-exported-variables = "export" +{ PR-exported-variable-declaration } .

GR-instance-comment = GR-comment .

B.6 Module body diagram

B.6.1 Body definition part

GR-body-definition-part =
GR-frame-symbol
IS ASSOCIATED WITH
GR-body-definition-label
CONTAINS (
GR-body-definition
[GR-body-comment]) .

GR-body-definition-label = < *text* > .

GR-body-definition =
[GR-declaration-part]
[GR-initialization-part]
[GR-transition-part] .

GR-body-comment = GR-comment .

B.6.2 Declaration part

GR-declaration-part =
GR-declaration-symbol
CONTAINS
{ GR-declarations } .

GR-declarations =
PR-constant-definition-part
| PR-type-definition-part
| PR-state-definition-part
| PR-state-set-definition-part
| PR-variable-declaration-part
| PR-procedure-and-function-declaration-part
| PR-channel-definition
| PR-module-variable-declaration-part .

See [ISO97] § 7.3 for the Extelle/PR syntax of each of the above declarations.

B.6.3 Initialisation part

GR-initialization-part =
GR-initialization-symbol

IS FOLLOWED BY
GR-transition-definition .

GR-initialization-symbol = 

.

B.6.4 Transition part

GR-transition-part = { GR-transition-definition } .

GR-transition-definition =
(GR-transition-condition-part | GR-transition-action-part) .

GR-transition-condition-part =
GR-condition
IS FOLLOWED BY
+{ GR-transition-condition-part | GR-transition-action-part } .

GR-condition =
GR-provided-clause
| GR-from-clause
| GR-to-clause
| GR-delay-clause
| GR-when-clause
| GR-priority-clause
| GR-any-clause .

GR-transition-action-part =
GR-transition-statement-part
IS FOLLOWED BY
[GR-to-clause] .

GR-transition-statement-part =
GR-frame-symbol

```
        CONTAINS (
            [ GR-transition-name ]
            [ GR-transition-declaration-part ]
            [ GR-transition-comment ]
            GR-statement-group ) .

GR-transition-name = PR-identifier .

GR-transition-declaration-part =
        GR-declaration-symbol
        CONTAINS (
            PR-constant-definition-part
            PR-type-definition-part
            PR-variable-declaration-part
            PR-procedure-and-function-declaration-part ) .

GR-transition-comment = GR-comment .

GR-statement-group =
        GR-statement
        IS FOLLOWED BY
            [ GR-statement-group ] .

GR-statement =
        GR-procedure-reference
        | GR-output
        | GR-pascal-code
        | GR-initialize
        | GR-terminate
        | GR-release
        | GR-attach
        | GR-detach
        | GR-connect
        | GR-disconnect
```

| GR-all-statement
| GR-forone-statement .

B.7 Transition condition clauses

B.7.1 Provided clause

GR-provided-clause =
GR-condition-symbol
CONTAINS (
PR-boolean-expression
| "otherwise") .

GR-condition-symbol = ⟨ ⟩ .

B.7.2 From clause

GR-from-clause =
GR-from-symbol
CONTAINS (
GR-from-expression .

GR-from-expression = GR-from-include-list | GR-from-exclude-list) .

GR-from-symbol = GR-from-state-symbol | GR-from-state-set-symbol .

GR-from-state-symbol =  .

GR-from-state-set-symbol =  .

GR-from-include-list = +{ GR-from-include-element } .

GR-from-include element = GR-state-identifier | GR-state-set-identifier | "*" .

GR-from-exclude-list = "*(" +{ PR-from-exclude-element } ")" .

GR-from-exclude-element = GR-state-identifier | GR-state-set-identifier .

GR-state-identifier = PR-identifier .

GR-state-set-identifier = PR-identifier .

B.7.3 To clause

GR-to-clause =
    GR-to-state-symbol
    CONTAINS
        GR-to-expression .

GR-to-state-symbol =  .

GR-to-expression =
    GR-state-identifier
    | "same"
    | "-" .

B.7.4 Delay clause

GR-delay-clause =
    GR-delay-symbol
    CONTAINS
        GR-delay-min-wait-time
        [GR-delay-max-wait-time | "*"] .

GR-delay-symbol =  .

GR-delay-min-wait-time = PR-expression .

GR-delay-max-wait-time = PR-expression .

B.7.5 When clause

GR-when-clause =
    GR-when-symbol
    CONTAINS
        PR-when-ip-reference .

GR-when-symbol =  .

B.7.6 Priority clause

GR-priority-clause =
    GR-priority-symbol
    CONTAINS
        PR-priority-constant .

GR-priority-symbol =  .

B.7.7 Any clause

GR-any-clause =
    GR-any-symbol
    CONTAINS
        PR-domain-list .

GR-any-symbol =  .

B.8 Transition action statements

B.8.1 Procedure reference

GR-procedure-reference =
    GR-procedure-symbol
    CONTAINS
        PR-procedure-statement .

GR-procedure-symbol =  .

B.8.2 Output

GR-output =
    GR-output-symbol
    CONTAINS
        PR-interaction-reference [PR-actual-parameter-list] .

GR-output-symbol =  .

B.8.3 Pascal code

GR-pascal-code =
    GR-text-symbol
    CONTAINS
        PR-statement-sequence .

GR-text-symbol =  .

B.8.4 Attach

GR-attach =
    GR-attach-symbol
    CONTAINS
        "attach" PR-external-ip "to" PR-child-external-ip .

GR-attach-symbol = GR-frame-symbol .

B.8.5 Detach

GR-detach =
    GR-detach-symbol
    CONTAINS
        "detach" (PR-external-ip | PR-child-external-ip | PR-module-variable) .

GR-detach-symbol = GR-frame-symbol .

B.8.6 Connect

GR-connect =
    GR-connect-symbol
    CONTAINS
        "connect" PR-connect-ip "to" PR-connect-ip .

GR-connect-symbol = GR-frame-symbol .

B.8.7 Disconnect

GR-disconnect =

GR-disconnect-symbol
CONTAINS
"disconnect" (PR-connect-ip | PR-module-variable) .

GR-disconnect-symbol = GR-frame-symbol .

B.8.8 Initialize

GR-initialize =
GR-initialize-symbol
CONTAINS
PR-module-variable
"with" PR-body-identifier ["(" PR-actual-module-parameter-list ")"] .

GR-initialize-symbol =  .

B.8.9 Terminate and Release

GR-terminate =
GR-terminate-symbol
CONTAINS
"terminate" PR-module-variable .

GR-release =
GR-frame-symbol
CONTAINS
"release" PR-module-variable .

GR-terminate-symbol = GR-frame-symbol .

GR-release-symbol = GR-frame-symbol .

B.8.10 All statement

GR-all-statement =
    GR-all-domain
    LOOPS THROUGH
        GR-statement-group .

GR-all-domain = "all" PR-domain-list .

B.8.11 Forone statement

GR-forone-statement =
    GR-forone-domain
    IS FOLLOWED BY (
        GR-forone-true-part
        GR-forone-otherwise-part) .

GR-forone-domain = "forone" PR-domain-list

GR-forone-true-part =
    GR-condition-symbol
    CONTAINS
        PR-boolean-expression
    IS FOLLOWED BY
        GR-true-statement-group .

GR-otherwise-statement-group = GR-statement-group .

GR-forone-otherwise-part =
    GR-condition-symbol
    CONTAINS
        "otherwise"
    IS FOLLOWED BY
        [GR-otherwise-statement-group].

GR-otherwise-statement-group = GR-statement-group .

Bibliography

[ABS+96] P Amer, G Burch, A Sethi, D Zhu, T Dzik, R Menell, and M Mcmahon. Estelle specification of MIL-STD-188-220A Datalink Layer. In *Proceedings of MILCOM '96*, October 1996.

[AH75] Igor Aleksander and F. Keith Hanna. *Automata Theory: An engineering approach.* Crane, Russak and Company, Inc, USA, 1975.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[BH89] Ferenc Belina and Dieter Hogrefe. The CCITT-Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16:311–341, 1988/1989.

[Boc78] G.V. Bochman. Finite state description of communication protocols. *Computer Networks*, 2:361–378, 1978.

[Bud92] S. Budkowski. Estelle development toolset EDT. *Computer Networks and ISDN Systems: Special Issue on Tools and Formal Techniques for Protocol Engineering*, 25(5):63–82, 1992.

[Chi94] Noel Chiappa. The TFTP protocol (revision 2). July 22, 1994.

[CJG88] J. Monin C. Jard and R. Groz. Development of Véda: A prototyping tool for distributed algorithms. *IEEE Transactions on Software Engineering*, 14(3), March 1988.

[Cou89] J.P. Courtiat. Estelle*, a powerful dialect of Estelle for OSI protocol description. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII.* North-Holland, 1989.

[Cou91] J.P. Courtiat. Introducing a rendez-vous mechanism in Estelle: Estelle*. In Quemada, editor, *Formal Description Techniques III.* North-Holland, 1991.

[EP97] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit.* IEEE, 1997.

[Hog89] D. Hogrefe. *Estelle, LOTOS und SDL.* Springer-Verlag, Berlin, 1989. In german.

[INT97] Institut National des Télécommunications. *X Window interface for EDT*, 1997.

[ISO83] *ISO/IEC 7185, Information Technology - Programming Languages - PASCAL.* International Standards Organisation, 1983.

[ISO94] *ISO/IEC 7498-1, Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* International Standards Organisation, 1994.

[ISO97] *ISO/IEC 9074(E), Estelle: A Formal Description Technique based on a finite state transition model.* International Standards Organisation, 1997.

[ITU93a] *MSC, Message Sequence Charts, ITU-T Recommendation Z.120.* International Telecommunications Union, Geneva, 1993.

[ITU93b] *SDL, Specification and Description Language, ITU-T Recommendation Z.100.* International Telecommunications Union, Geneva, 1993.

[Kri97] Pieter Kritzinger. *Introduction to Computer Networks.* Department of Computer Science, University of Cape Town, 5.1 edition, 1997.

[Neu95] P. Neumann. *Computer Related Risks.* Addison-Wesley Publishing Company, 1995.

[NP90] D. New and Amer P. Adding graphics and animation to Estelle. *Information and Software Technology*, 32(2), 1990.

[NP93] D. New and Amer P. Protocol visualisation in Estelle. *Computer Networks and ISDN Systems*, 25:741–760, 1993.

[oST92] United States National Institute of Standards and Technology. OSIKit Tools from NIST. Internet document available at ftp:osi.ncsl.nist.gov/pub/osikit, March 1992. For more information send e-mail to estelle@osi.ncsl.nist.gov.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Bonn, 1962.

[RAT96] Unified Modelling Language for real-time systems design. *Internet document - http://www.rational.com*, 1996.

[Sab97] JP Sabathe. Premier retour d'experience sur le noyau de verification formelle de l'outil ASA+. *Genie Logiciel*, (45), September 1997. In French.

[SCV93] A. Loureiro S. Chanson and S. Vuong. On tools supporting the use of Formal Description Techniques in protocol development. *Computer Networks and ISDN Systems*, 25(7), 1993.

[SL92] Rachid Sijelmassi and Richard J. Linn. Guidelines for using Estelle to specify OSI services and protocols. *Computer Networks and ISDN Systems*, 23, 1992.

[Sol81] K.R. Sollins. The TFTP protocol (revision 2) RFC 783. June, 1981.

[Som92] Ian Somerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[Tel98] Telelogic. *SDT Toolset*, 1998. http://www.telelogic.se.

[Ver89] Verilog SA, 150 rue Nicolas Vauquelin, 31106 Toulouse CEDEX, France. *Véda reference manual*, November 1989.

[Ver96] Verilog SA, 150 rue Nicolas Vauquelin, 31106 Toulouse CEDEX, France. *Object-GEODE: SDL Editor - User's Guide*, 1996.