NOTICE

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

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

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

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

AVIS

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

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

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

FULL IMPLEMENTATION OF A TEST DESIGN METHODOLOGY
FOR PROTOCOL TESTING

Vassilios Koukoulidis

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering at
Concordia University
Montréal, Québec, Canada

March, 1989

© Vassilios Koukoulidis, 1989
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L’auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L’auteur conserve la propriété du droit d’auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ABSTRACT

Full Implementation of a Test Design Methodology for Protocol Testing

Vassilios Koukoulidis

The implementation of a system for analysis of protocols is presented. This analysis is currently used for test sequence generation for protocol implementations. The protocol must be specified in Estelle, a protocol specification language based on an extended finite state machine model. The analysis first verifies that the specification is free of syntax and semantic errors. Next, a static analysis called normalization is applied. Normalization eliminates the transfer of control introduced by Estelle constructs such as procedure or function calls, conditionals, loop statements and state sets. Next, module merging is performed in order to eliminate intermodule communication in case of multi-module specifications. Intermodule communication is undesirable in black box testing, because it is not possible to observe internal interactions. The next step is the generation of information for graphical representation of data and control flow. Data flow models the operations applied on the input interaction parameters and context variables in order to determine the value of the output interaction parameters. Control flow models the finite state machine implemented by the Estelle specification. The output of the system can be used for generation of black box testing data. Test generation is demonstrated using a two module protocol specification and applying all the steps of processing until test sequences are produced.
ACKNOWLEDGEMENTS

Many people have contributed in the preparation of this thesis and the system described in it. I would like to thank my supervisor professor Behcet Sarlıkaya, who developed the theory implemented here and introduced it to me. He read thoroughly earlier drafts of this thesis and made useful suggestions and remarks that resulted in the current form. I would also like to thank the Department of Communications of Canada for providing financial support, Michel Barbeau for explaining to me the details and difficult points of a similar system developed by him thus helping to design a new more general implementation, and Rick To for programming the algorithm for merging finite state machines.

Last but not least I would like to thank my parents, Nicolas and Marla, and my brother, Yannis, for providing me financial and moral support, my fiancée Gina Cabadaldis for being so understanding and supporting during the last two years and my friend and colleague Vassils Zoukos for helping me to understand details of the field of protocols and distributed systems with his intriguing discussions.
Αφιερώνεται στους

Νίκο, Μαρία, Γιάννη και Τζίνα
# Table of Contents

LIST OF FIGURES ........................................................................................................... x

LIST OF TABLES ............................................................................................................... xl

CHAPTER 1: INTRODUCTION ...................................................................................... 1

CHAPTER 2: A TEST DESIGN METHODOLOGY ...................................................... 10

2.1 The Estelle Protocol Specification Language ...................................................... 10

2.2 Transformations and Control and Data Flow Information ................................. 14

2.3 Data and Control Flow Graphs and Protocol Functions ..................................... 16

2.4 Test Sequence Generation .................................................................................... 17

2.5 The Prolog Programming Language .................................................................... 17

CHAPTER 3: COMPILATION AND NORMALIZATION ............................................... 20

3.1 COMPILATION ....................................................................................................... 21

3.1.1 Syntax Tree Builder ......................................................................................... 25

3.1.2 Conversion of the Syntax Tree to a Prolog Term ........................................... 28

3.1.3 Translation of Symbol Table to Prolog .......................................................... 29

3.1.4 Performance of the Compiler Module ............................................................ 33

3.2 NORMALIZATION ................................................................................................. 35

3.2.1 Declaration Part Processing ............................................................................. 36

3.2.2 With Statements Replacement ....................................................................... 38
3.2.3 Procedure and Function Calls Replacement ................................ 39
   3.2.3.1 Procedure Calls .................................................................... 39
   3.2.3.1 Function Calls ..................................................................... 40
3.2.4 Conditional Statements Replacement ............................................ 44
3.2.5 For, All and While Statements Replacement ................................. 45
   3.2.5 Processing of From and To Clauses ......................................... 49
3.3 IMPLEMENTATION OF NORMALIZATION MODULE .......................... 50
   3.3.1 Performance of Normalization Module ...................................... 59
   3.3.1 Printing ............................................................................... 60
CHAPTER 4: DATA AND CONTROL FLOW ANALYSIS ............................... 62
4.1 DATA FLOW ANALYSIS .................................................................. 62
   4.1.1 Data Flow Analysis - First Phase ............................................ 63
      4.1.1.1 Declaration Part Processing .............................................. 63
      4.1.1.2 Transition Declaration Part Processing ............................. 68
      4.1.1.3 Implementation of First Phase of Data Flow
         Analysis ................................................................................ 73
      4.1.1.4 Performance of the First Phase of Data Flow
         Analysis ................................................................................ 76
   4.1.2 Data Flow Analysis - Second Phase ......................................... 76
      4.1.2.1 Implementation of Second Phase of Data Flow
         Analysis ................................................................................ 82
      4.1.2.2 Performance of the Second Phase of Data Flow
A.2.1 nf (Normal Form) Command ........................................ 112
A.2.2 cmbn (Combine Modules) Command ................................ 113
A.2.3 dtf (Data and Control Flow) Command ............................ 114
A.3 Warnings and Error Messages ......................................... 115
  A.3.1 Compilation Phase Warnings and Error Messages ............... 115
  A.3.2 nf Command Error Messages .................................... 115
  A.3.2 Unchecked Problems ................................ ........... 116
A.4 Glossary ....................................................................... 117

APPENDIX B: THE ALTERNATING BIT PROTOCOL ......................... 119
APPENDIX C: MERGED ALTERNATING BIT PROTOCOL ................... 124
APPENDIX D: TEST SEQUENCES FOR THE ALTERNATING BIT
  PROTOCOL ........................................................................ 129
List of Figures

Figure 1.1. The Open Systems Interconnection Model

Figure 1.2. Structure of an OSI layer

Figure 1.3. A test architecture

Figure 2.1. Structure of a system for processing of Estelle specifications

Figure 3.1. Structure of the compiler module

Figure 4.1. Data flow graph representation of reference to an array element

Figure 4.2. Data flow graph representation of ANY clauses

Figure 4.3. Control flow graph of the alternating bit protocol

Figure 6.1. Functions of the alternating bit protocol

Figure 6.2. Control flow graph of the merged alternating bit protocol
# List of Tables

Table 3.1. Performance of the compiler module

Table 3.2. Performance of the normalization module

Table 3.3. Performance of the printing module

Table 4.1. Performance of data flow analysis - phase I

Table 4.2. Performance of data flow analysis - Phase II

Table 4.3. Performance of control flow analysis

Table 5.1. Performance of module merging
CHAPTER 1

INTRODUCTION

A computer network is a collection of autonomous interconnected computers. The International Standards Organization (ISO) has modeled the structure of a computer network with the OSI (Open Systems Interconnection) Reference Model, [6] (figure 1.1). Any system designed for networking can follow OSI's layering approach. Each layer uses the services provided by the lower layer in order to offer certain services to the higher layer and so on (figure 1.2). The services provided by each layer are explicitly specified by the ISO. The details of implementation of a layer are shielded from the layer's service user. Therefore, layer N on one machine can carry on a conversation with layer N on the remote machine abstracting from the details of how this conversation is carried through by the service provider (layer N-1). A protocol is the set of conventions and rules used in the conversation between corresponding layers (also called peer processes), [24]. The data exchanged between peer processes are carried with protocol data units (PDUs).

There exists a large variety of protocol implementations from different manufacturers running on different machines implementing the services of each layer of the OSI model. In such an heterogeneous environment the most practical way of proving that networking can be achieved is testing the various implementations, [21]. This thesis focuses on a type of testing called conformance testing. The aim of conformance testing is to check whether an implementation conforms to the protocol standard defined by ISO. Conformance testing can be single-layer or multi-layer. Multi-layer testing is out of the scope of this thesis.
Figure 1.1. The Open Systems Interconnection Model

Figure 1.3 shows an architecture (known as distributed single layer architecture, [14]) which can be used to test N-layer OSI protocols. The upper tester is a task using the N-layer service provided by the implementation under test (IUT). The lower tester is a task using (N-1)-layer service and resides in a remote computer. The lower and upper tester are coordinated (using the test coordination procedures) in order to stimulate the IUT with a given sequence of input interactions and observe the IUT's outputs. This sequence of interactions will be referred to as test sequence and is made up of a number of protocol data units (PDUs) sent to or received by the IUT.

A methodology for test sequence generation has been introduced by Sarikaya
Figure 1.2. Structure of an OSI layer

Figure 1.3. A test architecture
et al. in [10]. This methodology considers the IUT as a black box and assumes that a formal specification of the protocol implemented by the IUT is available. Estelle, [8], has been chosen as the formal specification language. Estelle is based on an extended finite state machine model which associates each transition with a number of actions, [11]. Sarikaya et al. propose a number of transformations that simplify an Estelle specification. These transformations are referred to as normalization. They eliminate control paths and procedure/function calls, and merge the modules of a multi-module specification to a single module using symbolic execution, [5]. The normalized specification is further processed to generate control and data flow graphs. The graphs are decomposed into subtours (i.e., sequences of transitions starting from and ending at the initial state) and protocol functions, respectively. Then test sequences are designed considering the values of input parameters and determining the corresponding outputs.

The objective of this thesis is to implement a system that normalizes and performs module merging on an Estelle specification and then produces control and data flow information. The output can be used by graphics tools developed earlier in order to display control and data flow graphs and design test sequences.

The thesis work is based on an earlier work done in 1987, [2]. That system covered a limited subset of an older dialect of Estelle. The implementation proposed by this thesis accepts an Estelle specification as defined by the newest Estelle standard. Additionally, it performs transformations on more Estelle constructs such as while and for loops, with and case statements and variant records and array data types. Also, the module merging routines have been added. Another important difference with the older system is the adaptation of an Estelle compiler written entirely in C, while the older system used Prolog for implementation of semantic analysis. The new approach increases the speed of compilation dramatically. These improvements make possible the analysis of real
protocols such as LAP-D, file transfer access and management (FTAM) or transport protocols. According to performance measurements included in the chapters to follow, the time needed to process totally a protocol specification ranges from a few seconds for small examples up to 40 minutes for large real-life protocols. It is not possible to make performance comparisons with the older system for two reasons: several of the older system's modules were implemented using a Prolog interpreter, while the newer system uses a Prolog compiler, and the Estelle dialect used by the older system is incompatible with the new Estelle. To the best of our knowledge there exists no other similar system with which performance comparisons could be made.

The system consists of a compiler which verifies that the specification does not contain syntactic or semantic errors and generates an intermediate form suitable for processing by the normalization, control and data flow analysis and module merging routines. The intermediate form is the specification's syntax tree accompanied with a symbol table containing information about variable and data type declarations. The compiler is implemented under the UNIX operating system and it uses the UNIX tools YACC and LEX for syntactic and lexical analysis and the language C for semantic analysis and intermediate form generation.

Normalization, control and data flow, and module merging are written in Prolog. The dialect used is Quintus Prolog, [28], which is compatible with DEC-10 Prolog, a rather standard Prolog dialect. Programming in Prolog consists of defining objects and rules describing the logic relationships between the objects, and asking questions about the objects and their relationships, [4]. The advantages of using Prolog instead of a conventional language for building a prototype can be summarized as follows, [25]:

- Less development time is required because the declarative style of Prolog
facilitates the transfer of a problem specification into a program.

- The likelihood of bug producing errors is reduced. The closeness of problem specification and program makes the correctness of Prolog code easily apparent. Errors can be easily detected because of the declarative style and modularity of Prolog.

- Prolog programs can be easily modified and maintained, since Prolog clauses are small self-contained units directly related to the specification of the application.

Lisp is similar to Prolog and could also be chosen as the implementation language. The reason for choosing Prolog instead of Lisp was that during the development period we had better Prolog tools (i.e. a very efficient compiler) and more working experience on Prolog.

The material in this thesis is organized as follows.

Chapter two gives an overview of the test design methodology introduced and describes briefly the Estelle and Prolog languages.

Chapter three describes in detail the compiler and normalization units. The lexical, syntactic and semantic analysis and error handling phases of the compiler were based on a prototype Estelle compiler developed by the National Bureau of Standards and enhanced by adding Pascal sets as explained in section 3.1. A new phase that generates the syntax tree and symbol table was designed and implemented as part of this thesis work (sections 3.1.1 - 3.1.3). Section 3.1.4 gives performance measurements of the compiler module. The theory of normalization was developed by Sarikaya et al. in references [19] and [18] and is discussed in detail in sections 3.2.1 - 3.2.6. The theory is accompanied with a plethora of examples illustrating the various transformations applied on the input Estelle specification. A contribution of this thesis is the implementation of the
normalization theory and is described in section 3.3. The routines that perform the normalization transformations are explained. Section 3.3.1 discusses a utility program that produces an Estelle specification from its Prolog syntax tree. All the examples of normalized Estelle code are output from this utility. Section 3.3.2 discusses the performance of the implementation.

Chapter four explains the generation of data and control flow information. The transformations were based on background theory developed by Sarıkaya in [19]. The scheme for explicitly identifying PDUs is similar to the scheme followed by Barbeau in [2]. The major contributions of this part of the thesis are the use of variant records for declaration of PDUs, the processing of primitive procedures/functions with parameters referring to PDUs, the expansion of a transition if the input PDUs cannot be identified from the provided clause, the processing of buffers keeping more than one kind of PDU, special handling of arrays and any clauses and the implementation of data flow analysis. Section 4.1 describes the two phases of data flow analysis, the implementation of each phase and their performance. In section 4.2 extraction of control flow information from an Estelle specification and its implementation and performance are discussed.

Chapter five gives a module merging algorithm and discusses its implementation. An earlier version of this algorithm for finite state machines was introduced by Sarıkaya and Bochmann in [15]. This thesis extends the earlier version to extended finite state machines and explains how transitions of interacting modules are merged (section 5.1). Section 5.2 gives the new version of the module merging algorithm and section 5.3 discusses its implementation. Performance is addressed in section 5.4.

Chapter six demonstrates with an example how test sequences can be generated starting from the protocol specification. The protocol used is the alternat-
ing bit protocol with two modules. The specification is normalized and passed through the first phase of data flow analysis. Next the two modules are merged and information for data and control flow graphs is produced. The data flow graph is displayed using an already developed program and partitioned into protocol functions. For each of those functions a number of subtours of the control flow graph is generated, each subtour defining a test sequence. The programs for subtour and test sequence generation were developed earlier, [23], and are not covered in detail by this thesis.

Chapter seven states the conclusions of this thesis.

The user's guide of the system is given in appendix A. It contains the commands to which the system listens, the list of error messages, and a glossary of the terms appearing during the interaction with the user.

Appendix B contains a specification in Estelle (the alternating bit protocol) which is used in chapter six for test sequence generation.

Appendix C gives the output of application of module merging on the alternating bit protocol.

Appendix D enumerates the test sequences derived for each function of the alternating bit protocol.

The examples used in the text are extracted from a simple transport protocol and the alternating bit protocol given in references [3] and [11], respectively. The example used in chapter six was adopted from reference [2]. LAP-D and FTAM single module protocols were available during the system development and used for performance evaluation of the various system units, as well as the single module alternating bit and two module transport protocols from references [11] and [3], respectively. All the performance experiments were conducted on a SUN 3/60 under UNIX 4.2 BSD operating system. The time required by the Prolog
programs was measured using Prolog's built in routine \textit{statistics/0}. The programs executed directly in UNIX shell were timed using the UNIX command \textit{time}.
CHAPTER 2

A TEST DESIGN METHODOLOGY

This chapter outlines a methodology for discovering errors in a protocol implementation. Reference [16] covers the theoretical background of the test design methodology and reference [23] describes its implementation. This methodology is inspired from functional program testing, [7], which views programs as collections of functions which are synthesized from other functions. Faults in synthesis of those functions result in program faults. Functional program testing has been proved applicable to protocol testing, [16]. The methodology is applied on Estelle specifications and consists of three steps:

a. transformation of the original specification and generation of control and data flow information,

b. data and control flow graph generation and determination of protocol functions and
c. generation of test sequences.

The first section of this chapter overviews the protocol specification language Estelle. Next, the test design methodology is outlined and finally the Prolog programming language is introduced. Prolog was used to implement the test design methodology.

2.1. The Estelle Protocol Specification Language

Modeling realistic protocols with finite state machines (FSMs) results in an immense number of states. Since many of these states are similar in terms of what the machine expects or responds to upon an interaction, it is possible to
combine them to a parametrized state (also called major state) thus reducing the state space size. This idea leads to the extended finite state machines (EFSMs) which contain states with variables. Actions on these variables can be performed when a transition from one state to another occurs. Estelle (Extended State Transition Language) is a protocol specification language based on the extended finite state machine model.

An Estelle specification consists of modules which communicate over channels connected to interaction points of modules. The communication over the channels is achieved through queues. The generic form of a channel definition is

```
channel channel_identifier (role_1, role_2);

by role_1:

interaction_1_1(parameter_list_1_1);
interaction_1_2(parameter_list_1_2);

...

by role_2:

interaction_2_1(parameter_list_2_1);
interaction_1_2(parameter_list_2_2);

...
```

Roles 'role_1' and 'role_2' are used to identify the role of a module communicating over this channel. The module may output only the interactions related to its role. Each module contains a number of transition rules explaining when a transition from a state is fired, what actions are performed during the transition and what the destination state is. A transition in Estelle has the following form:
trans

any v1:type_1, v2:type_2, ... do { for any value }
from state_1 { current state }
to state_2 { next state }
when lp_id.event { input event }
provided predicate { boolean expression }
priority expression { priority of the transition }
delay(min, max) { timing criterion in spontaneous transitions }
begin
...
{ transition block }
end

The any clause indicates that the transition can be executed for each possible permutation of the values of the variables v1, v2, ... Variables v1, v2, ... (i.e. the domain list) can be of ordinal type only. The from and to clauses specify the current and next state if the transition occurs. The when clause shows which event (i.e. interaction or service primitive) at the interaction point lp_id results in firing the transition. The provided clause contains a predicate on the parameters of the input event and/or the module variables (context variables). The priority clause can be used to determine which transition from a certain state should be executed first. The delay clause specifies timing criteria in spontaneous transitions (i.e. transitions without when). The transition is delayed for a time period 'min'. After this period it may be selected unless another transition is eligible. When 'max' period elapses the transition must be selected. It is possible that 'min' equals 'max'. The begin - end block describes in Pascal the actions taken when the transition fires. The variables used in a transition body
are referred to as context variables.

Two new statements introduced in Estelle are the statements **output** and **all**. **Output** is used to express outputs to other modules and its form is

```plaintext
output lp_id.event(expression1, expression2, ...);
```

Expression1, expression2, ... gives values to the parameters of the event. The general form of **all** statement is

```plaintext
all operand_list do statement;
```

The operand_list can declare variables referring to modules or Pascal variables. This thesis transforms **all** statements whose operand_list refers to Pascal variables only. In this case, the **all** statement is structured as

```plaintext
all v1:type_1, v2:type_2, ... do statement;
```

**All** iterates over the domains of v1, v2, ... until all possible permutations are covered, but unlike **for** statements the order of iterations is nondeterministic.

Estelle supports nondeterminism which is expressed by spontaneous transitions and more than one transition from a given major state for the same input event.

Abstractness of the specification can be achieved by declaring some data types incompletely using the three-dot notation (e.g. `buffer_type = ...`) or by declaring procedure and/or functions as primitive.

Estelle reserved words appear in **bold** when used inside text. Program fragments and names of variables, data types, procedures and functions are printed in **italics**.
2.2. Transformations and Control and Data Flow Information

Figure 2.1 outlines the structure of a system implementing the first step of the methodology mentioned. The *compilation* module verifies that the Estelle specification is free of syntactic and semantic errors and produces an intermediate form of the specification easily processable by the other modules. The transformations are carried out by the modules *normalization* and *data flow analysis - phase I* and simplify the determination of data and control flow graphs. Normalization removes from the input specification all the Estelle constructs that introduce paths or transfer of control during the execution of a transition (e.g. if statements, *procedure* calls, etc.). Data flow analysis consists of two phases. Application of the first phase on a normalized specification identifies the kind of PDUs exchanged in interactions or referred to by variables. This information is reflected in the names of variables or interactions related to PDUs and, subsequently, on their declarations. The result is an equivalent Estelle specification which is going to be used by the remaining steps of the test design methodology and whose transitions are called *normal form transitions*. Since all the transformations are applied on this intermediate form, a printing module is necessary to get back to an Estelle representation of our specification. The normalized specification can be edited by the user (e.g. when the kind of PDUs cannot be determined from the context) and submitted to the module merging routines. These routines produce a single module specification (If there is more than one module). The output is passed to the *control flow analysis* and *data flow analysis - phase II* in order to produce data for control and data flow graph generation (programs *cgtool* and *dfgtool* respectively, [23]). The system of figure 2.1 is the subject of this thesis and is covered in detail in chapters 3, 4 and 5.
Figure 2.1. Structure of a system processing Estelle specifications
2.3. Data and Control Flow Graphs and Protocol Functions

A data flow graph models the manipulations performed on the parameters of an input interaction or context variables in order to determine the values of output interaction parameters. The data flow graph consists of four types of nodes $I$, $D$, $F$ and $O$ nodes representing input primitives, data, operations on data and output primitives respectively. These nodes are connected with arcs indicating the flow of data from the data source (e.g., I-nodes) to the data sinks (e.g., O-nodes). If an operation (F-node) uses parameters called by value (D-nodes) the arcs are directed from the D-node towards the F-node. If the parameters are called by reference their corresponding D-nodes are connected to the F-node by a bidirectional arc. The form of a data flow graph is described in detail in [16] and [23].

In order to derive protocol functions it is necessary to partition the data flow graph into blocks, each block representing the flow over a single context variable (modeled by a D-node) or an O-node (when the O-node is assigned directly by an I-node or an F-node). Reference [16] describes an algorithm that partitions the data flow graph into such blocks and [23] demonstrates the algorithm's implementation in a tool named dfgtool. This refinement of a data flow graph usually produces a large number of blocks. Several of these blocks can be combined to produce a functional block. A functional block corresponds to a protocol function. A careful functional decomposition of the data flow graph results in functional blocks with minimum communication (i.e. data flow) among them. The merging of elementary blocks in order to produce functional blocks cannot be fully automated because it is not possible to determine automatically what variables were used to form a protocol function. Some automation can be obtained by merging blocks whose O-nodes are of the same data type. Also, if the I-nodes of one block are contained in the O-nodes of another block the two blocks can be
merged. *Dfgtool* provides the user the ability to compose protocol functions from the data flow graph interactively.

Information about major state changes is excluded from the data flow graph. A *control flow graph* models the transitions from one major state to another. A tool developed to display the control flow graph (*cgtool*) is presented in [23].

### 2.4. Test Sequence Generation

A *transition tour* over the control flow graph is a sequence of transitions starting from the initial state and covering all possible transitions in the protocol specification. Any subsequence of the transition tour starting from and ending at the initial state is called a *subtour*. Each functional block of the data flow graph is associated with one or more subtours so that each arc in the block is covered at least once. The set of subtours derived for each functional block is a test sequence for the corresponding protocol function. The normal form transitions composing each subtour specify the behavior of the protocol if this subtour is followed. Any deviation from this behavior indicates an error.

### 2.5. The Prolog Programming Language

Prolog is a programming language based on predicate logic: Given a number of *facts* and *rules* over these facts, one can ask *queries*. The constructs of Prolog that model the facts and rules are called *clauses*. A query constitutes a *goal*. Syntactically, a clause comprises a head and a body. The head is a boolean term and the body a sequence of zero or more goals. Generally, a clause can be written as

\[
<\text{head}> \leftarrow <\text{goal}1>, <\text{goal}2>, \ldots
\]

A Prolog program consists of clauses. For example the clauses that implement the concatenation of two lists are:
append([], X, X).

append([X|X], Y, [X|Z]) :- append(X, Y, Z).

In these clauses we can see one of the most frequently used Prolog structures, the list. The special symbol [] represents an empty list. The notation [X|Y] represents a list whose first element or head is X and the list of the rest elements or tail is Y. Thus the append/3 clauses give a recursive rule on appending two lists (second clause) and a fact on the result of appending a list to the empty list (first clause). The first clause is used as the condition that ends the recursion.

A logical variable is the means of asking queries that can give an unknown answer or more than one answer. A variable can either be instantiated to an object or not and after instantiation it can refer only to that object.

Here is how Prolog tries to answer a query (i.e. to satisfy a goal). When a goal is set Prolog searches the set of clauses for the first clause whose head matches or unifies with the goal. That is, the arguments of the clause match the arguments of the goal. Next, the clause is activated and its goals (if any) are executed from left to right. If Prolog fails to satisfy a goal, it backtracks rejecting the most recently activated clause and undoing all substitutions made when the clause and the goal matched. The subsequent clauses are searched in order to find another clause that matches the goal.

As explained in the introduction, Prolog is suitable for developing prototypes. The main reasons are Prolog's declarative style and modularity, which produce almost self-documentation easy-to-debug programs. New clauses can be easily inserted and modifications of older clauses require changes to small program units only.

Prolog has been used in a number of software and protocol applications. Reference [25] describes the use of Prolog to build a compiler. An interpreter for
LOTOS, another protocol specification language, was built using Prolog as explained in [26]. The use of Prolog for expressing and testing protocol specifications is explored in [27].

Throughout this thesis, Prolog routine names are together with the number of routine's arguments (e.g. append/3) because it is possible that one routine has more than one definition, each definition having a different number of arguments.
CHAPTER 3

COMPILATION AND NORMALIZATION

This chapter is concerned with three modules of the system being described:

a. compiler module, which transforms an input Estelle specification to a form suitable for processing by Prolog,

b. normalization module, which performs symbolic execution on the input Estelle specification producing an equivalent specification, and

c. printing module, which prints an Estelle specification from a tree representation produced by the normalization module.

The printing module has a general structure and it can print any Estelle construct from its tree representation. It is also used by the data flow analysis and module merging module.

3.1. COMPILATION

The input Estelle specification must be represented as a Prolog term in order to be efficiently manipulated by the normalization module. For example, the assignment statement

$$
credit := 0;
$$

can be translated to the Prolog term

$$
stmt(vrAcc(id(credit,149)), xpr(unsgndInt(0))).
$$

This term reads as
access the variable whose identifier is credit and assign to this variable an expression consisting of unsigned integer 0.

This representation is an alternative way of defining a tree in Prolog, [25]. The root of this tree is stmt. Names stmt, vrAcc, id, xpr and unsynldInt correspond to statement, variable access, identifier, expression and unsigned integer, respectively. Number 149 in this term corresponds to line number and is used during the processing of an Estelle specification for making error messages.

The representation of an Estelle specification as a Prolog tree must be accompanied by a data structure containing information frequently asked by normalization, data flow and control flow analysis modules. This information refers to various Estelle structures and types (e.g. channels or Interaction points) and is stored in a user defined Prolog structure named dictionary. The dictionary is a sorted tree. Each node of this tree is labeled with the name of an Estelle declaration and contains a field with the name's definition (for more details see 3.1.3). Clearly, searching a sorted tree is by far faster than searching an unsorted tree of declarations.

Given an Estelle specification, the compiler module produces a Prolog term and a dictionary. It also verifies that the input specification is syntactically and semantically correct. The structure of the compiler is drawn in figure 3.1. The implementation of modules for lexical, syntax and semantic analysis, and error recovery is based on the Estelle compiler developed by the National Bureau of Standards (NBS) and they are discussed in detail in [12]. The NBS Estelle compiler translates an Estelle specification to a C code program, [13]. In the system being described by this thesis, the C code generation functions were removed thus reducing the task of the compiler to syntax and semantic checking only. Furthermore, the compiler was improved by adding Pascal sets and debugging the
Figure 3.1. Structure of the compiler module
functions for semantic checking of arrays of interaction points.† The modules that are common to our system and the NBS compiler will be presented briefly. The module building the parse tree as a Prolog term and the dictionary is presented in the sequel.

**Lexical analysis** module reads *tokens* from the source specification and interacts with the parser (syntax analyzer) returning a *token code* or a *token value*. A token can be an Estelle reserved word or character literal (e.g. +, -, *, /), an identifier, a number (integer or real), or a character string. A token code is an integer assigned to each reserved word or character literal. A token value is a pointer to the area where an identifier, number or string is stored. This module is implemented using the UNIX utility LEX, [9].

**Syntactic analysis** module (parser) is produced by compiler generator YACC (a tool of UNIX, [9]). All the actions performed by the compiler (i.e. lexical, syntax and semantic analysis, error handling, Prolog tree and dictionary construction) are invoked from the parser. The parser builds two trees concurrently: one to be used for syntax and semantic checking and one to be translated to a Prolog term. The first tree is built in fragments. Each fragment corresponds to a block of Estelle code (e.g. a transition block, a procedure/function declaration or a body) and it is destroyed when this block ends. The second tree is the global syntax tree and is created by an independent module (Prolog Tree & Dictionary Construction module in figure 3.1) which interacts with the parser without affecting the rest of the parser's actions. Since all the other modules are called as part of parsing, the translation of the input Estelle specification is performed in a single pass.

† The problem occurred in the declarations of arrays of interaction points whose base type was a subrange type with undefined low and high bounds (i.e. with low and high bounds declared as '...').
Error handling is concerned with resynchronizing the parsing and performing some actions when an error occurs (i.e. recovering from errors). For this purpose some alternative grammar rules are specified in YACC source code. They use YACC's token error with a literal (e.g. a semicolon) or with the nonterminal symbol ResynchToken (a symbol is nonterminal if it is defined as another rule).

Semantic analysis collects type information and verifies that operators and operands are used consistently within expressions and statements. It also performs type coercions when this is permitted by Estelle ([12], [1]). All the information concerning symbols declared in the Estelle specification (e.g. procedures, variables, etc) is stored in the symbol table. The symbol table is implemented as a hash table consisting of a fixed array of pointers to symbol table entries. The index into this fixed array is obtained by hashing on the identifier's name. In case of conflict, the new entry is linked to the previous one. In other words each pointer in the array points to a linked list of symbol table entries. Semantic analysis is invoked from the actions part of the YACC source and is written in C.

A new feature added to the NBS compiler is Pascal sets. In the occurrence of a set declaration, a new symbol table entry is created and a new node is linked to the Prolog tree:

StructuredType :

```
| SET OF SimpleType
{ Syntax;
  $$ = newtype(SET_T, $$);
  mktree(4, 1, S_NULL, S_NULL, S_NULL, S_NULL, S_NULL, S_NULL, S_NULL);
};
```

where *SimpleType* is the base type of the set (for more details on function *mktree()* see 3.1.1). This extension allows the use of sets in normalization, data
flow and control flow analysis despite the fact that the NBS Estelle compiler does not handle them.

3.1.1. Syntax Tree Builder

The syntactic and semantic analysis routines (YACC source code) build fragments of the syntax tree which are destroyed each time a block is exited. In order to build a complete syntax tree without affecting the compilation phase, a stack is created in parallel with the stack created by YACC. The routines which make this stack are inserted in the actions part of the YACC source code.

The syntax tree is built bottom-up as follows. Each time a grammar rule is encountered a new parse tree node is created in the following way: If the body of this rule consists of terminal and/or nonterminal symbols, a new node for each one of them is created and pushed into the stack. Each one of the nonterminal symbols corresponds to a new rule, which is treated in the same way. Therefore, a rule is completely recognized (i.e. reduced) when all of its nonterminals are reduced. Tree node creation is an action associated with each reduction. When all nonterminals of a rule are reduced, their corresponding nodes are popped out of the stack and linked — possibly with terminals, if they exist — in order to constitute the node of the parent rule. This process continues until we reach the root node (the specification header). Consider, for example, the first grammar rule of the Estelle compiler

```plaintext
Specification : */ ref: %Start *//* empty */ /* so null file won't cause errors */

SPECIFICATION IDENTIFIER SystemClass ','
{ Syntax; specdecl($2, $3); }
DefaultOptions
TimeOptions
BodyDefinition
END ','
{ Syntax;
endbody(TRUE);
mktree(1, 5, $2, S_NULL, S_NULL, S_NULL, S_NULL, S_NULL, S_NULL);

```
The parent node is Specification (this node also happens to be the root of the complete syntax tree). The statement mktree in rule's action part refers to the node creation. When all the subtrees of the parent node (i.e. SystemClass, DefaultOptions, TimeOptions and BodyDefinition) are constructed mktree() (make tree) is called. The first argument (Integer 1) specifies the node type and it is used for distinction of rules with the same header (i.e. for rules with more than one production). The second argument (integer 2) represents the number of subtrees. The third argument is the name of the specification (i.e. the value of IDENTIFIER); this is a terminal symbol and also hangs from the root. The remaining four subtrees are popped from the stack (the stack's structure will be shown later in this chapter) and not given as arguments. Function prtree() is called when the syntax tree is completely built. It pops the last pointer remaining in the stack (i.e. the pointer to the root) and invokes the functions which print the syntax tree as a Prolog term (see 3.1.2).

The structure that defines a node of the syntax tree is a recursive one:

```c
typedef struct stnode * STNODEPTR;
struct stnode {
    int     stn_type;
    int     stn_line;
    STNODEPTR stn_fields[6];
};
```

Stack building routines are similar to the ones described in [10]. The definition of stack data type is

```c
#define MAXSTACK 256
STNODEPTR stack[MAXSTACK];
int sp = 0;
```
Function \texttt{mktree()} builds the syntax tree from its nodes. This function uses
the stack and is defined as

\begin{verbatim}
mktree(type, nfields, string0,string1,string2,string3,string4,string5)
int type, nfields;
STR string0, string1, string2, string3, string4, string5;
{
    unsigned size;
    STNODEPTR p;

    if (nfields != 0) {
        size = sizeof(*p) + (nfields - 1)*sizeof(STNODEPTR);
        if ((p = (STNODEPTR) CALLOC(1, size)) == STN_NULL)
            usererror("(mktree) ran out of memory");

        p->stn_type = type;
        p->stn_line = yylineno;

        switch (nfields)
        {
            default: cerror("tree: nfields=%d", nfields);
            case 6:
                p->stn_fields[5] =
                    (string5 == S_NULL) ? pop() : (STNODEPTR)string5;
            case 5:
                p->stn_fields[4] =
                    (string4 == S_NULL) ? pop() : (STNODEPTR)string4;
            case 4:
                p->stn_fields[3] =
                    (string3 == S_NULL) ? pop() : (STNODEPTR)string3;
            case 3:
                p->stn_fields[2] =
                    (string2 == S_NULL) ? pop() : (STNODEPTR)string2;
            case 2:
                p->stn_fields[1] =
                    (string1 == S_NULL) ? pop() : (STNODEPTR)string1;
            case 1:
                p->stn_fields[0] =
                    (string0 == S_NULL) ? pop() : (STNODEPTR)string0;
            }
            push(p);
        }
        else push(STN_NULL);
    }
\end{verbatim}

Notice that the arguments of \texttt{mktree()} are string pointers which represent the
names of identifiers. These pointers are cast to \texttt{STNODEPTR} type and hang
directly from the tree. If the value of formal parameter \texttt{nfields} is greater than
the number of arguments (i.e. not all the children nodes are given) the missing nodes are popped from the stack.

3.1.2. Conversion of the Syntax Tree to a Prolog Term

When the syntax tree is completed and if the specification does not contain errors, function prtree() passes its root to function spc() (3.1.1):

```c
VOID prtree()
{
    prolog_tree = OpenWrite("estelle.tree");
    FPRINTF(prolog_tree, " %d.
", nerrs + nsynerrs);
    if ((nerrs + nsynerrs) == 0) {
        FPRINTF(prolog_tree, " spc");
        spc(pop());
        FPRINTF(prolog_tree, ".\n");
    }
    CloseFile(prolog_tree);
}
```

Then spc() fires the tree printing as a Prolog term top-down. There is one function for each node type. The initial function, which prints the specification header and all the subtrees hanging off, is

```c
VOID spc(n)
STNODEPTR n;
{
    if (n != STN_NULL)
    {
        FPRINTF(prolog_tree,
            " (id(%s,%d), tolow((STR)n->stn_fields[0]), n->stn_line);
            FPRINTF(prolog_tree, " sstmClass");
            sstmClass(n->stn_fields[1]);
            FPRINTF(prolog_tree, " ,dfltOpt");
            dfltOpt(n->stn_fields[2]);
            FPRINTF(prolog_tree, " ,tmOpt");
            tmOpt(n->stn_fields[3]);
            FPRINTF(prolog_tree, " ,bdDf");
            bdDf(n->stn_fields[4]);
            FPRINTF(prolog_tree, ")");
    }
}
```

The first statement inside if's body prints specification's name. The rest prints
the subtrees for system class, default options, time options and body definition. The structural similarity of this function with the first rule of Estelle compiler (see 3.1.1.) is obvious. If the root node is not null, similar functions are called to print the hanging nodes of the syntax tree. $Spc()$ is called when the starting rule of YACC source code is reduced. For example, the tree produced for specification

```
**specification Example systemprocess;**
timescale seconds;
end.
```

is

```
spc(
    id(example,8),
sstmClass(systemprocess),
dfltOpt,
tmOpt(id(seconds,8)),
bdDf(dclPrt,initPrt,trDc!Prt)
).
```

### 3.1.3. Translation of Symbol Table to Prolog

The symbol table provides efficient representation of specification declarations throughout the syntactic and semantic analysis phase. This information is necessary during the normalization and data and control flow analysis phase. More specifically, the normalization and data and control flow analysis phases require information concerning

- procedures, functions and variables,
- interaction points,
- channels and
- data types

in order to perform symbolic replacement and define the input, output, function and data nodes in the data flow analysis graph.
A convenient data type for symbol table representation in Prolog is the dictionary, [25]. A dictionary is defined recursively. Thus

\[ \text{dic}(\langle \text{name} \rangle, \langle \text{value} \rangle, \langle \text{dic-1} \rangle, \langle \text{dic-2} \rangle) \]

pairs \( \langle \text{name} \rangle \) with \( \langle \text{value} \rangle \), where \( \langle \text{dic-1} \rangle \) and \( \langle \text{dic-2} \rangle \) are subdictionaries. The dictionary is also ordered alphabetically with respect to \( \langle \text{name} \rangle \). That is, all names in \( \langle \text{dic-1} \rangle \) (\( \langle \text{dic-2} \rangle \)) precede (succeed) \( \langle \text{name} \rangle \).

In order to obtain this dictionary the symbol table created during the syntactic and semantic analysis phase is printed as a sequence of Prolog lists of the form

\[ [\langle \text{name} \rangle, [\langle \text{class} \rangle, [\langle \text{definition} \rangle]]] \].

The \( \langle \text{name} \rangle \), \( \langle \text{class} \rangle \) and \( \langle \text{definition} \rangle \) values are obtained from the symbol table definitions for each identifier. Each identifier corresponds to a node structure which contains pointers to substructures hanging off this identifier (e.g. fields, parameters etc.). This representation dictates a tree structure for the dictionary entries which is realized using the list form shown above. Identifiers declared within the scope of another identifier are represented as nested lists. If \( \langle \text{class} \rangle \) is function, procedure or module header this list is followed by a sequence of parameters which are appended to the function, procedure or module body definition. The second element of the above list (original or modified) constitutes the \( \langle \text{value} \rangle \) element of a dictionary entry. Consider, for example, the symbol table entry of procedure \text{format_ack} in [11] and its conversion to a Prolog list:

\begin{verbatim}
block_begin.
[format_ack,[procedure,[]]].
[msg,[parameter,[msg_type]]].
[b,[parameter,[varndata_type]]].
param_list_end.
\end{verbatim}
This sequence of lists is read by Prolog clause `mk_dic` which creates a dictionary entry

```prolog
format_ack,
[[procedure,[]],[[msg,[parameter,[msg_type]]],[b,[parameter,[var,ndata_type]]]]].
```

If this entry were the only one, Prolog dictionary would look like

```prolog
dic(format_ack,
    [[procedure,[]],[[msg,[parameter,[msg_type]]],[b,[parameter,[var,ndata_type]]]]],
    void,
    void)
```

where `void` denotes an empty dictionary.

The body of the function converting the symbol table to Prolog lists has the structure

```prolog
sym2dic(p, entry_end, unique_name)
IDPTR p; /* identifier pointer */
BOOL entry_end; /* true if this entry should be closed with "." and "\n" */
BOOL unique_name;
{
    . . . /* local declarations */

    /* identifier name */
    FPRINTF(dic, "\%s\", unique_name ? p->id_pname : p->id_name);

    /* identifier class: */
    FPRINTF(dic,"\%s\", nameof[p->id_class]);

    /* identifier type */
    switch(p->id_class)
    {
        . . .

        case MODULE_ID:
            showtype(p->id_type, 1);
            FPRINTF(dic,"\n"); /* close header entry */
            /* show parameter list and interaction points */
            /* 1. parameter list */
            rlevel++;
            for( i = p->id_type->t_low; i != ID_NULL; i = i->id_list) {
                for (n = 1; n <= rlevel; n++)
                {
                    sym2dic(i, 1, 0);
                }
            } /* 2. interaction points */
            for( i = p->id_type->t_list; i != ID_NULL; i = i->id_list) {
                for (n = 1; n <= rlevel; n++)
                { sym2dic(i, 1, 0);
```


where "..." denote parts of code structured similarly to the code shown. The body of sym2dic() is mainly a switch statement over the type of each Estelle construct (e.g. module, procedure, etc.). This construct corresponds to a symbol table entry p whose class field id_class constitutes the switch's expression.

Prolog clause mk_dic/1 is responsible for building the dictionary from the lists created by function sym2dic(). This clause reads these lists one by one and inserts them in the dictionary except when the list corresponds to a procedure, function, or module header. In this case a list of parameters follows which should be appended in the dictionary entry (clauses app_par). If a parameter is defined outside the scope of a block_begin - block_end it is ignored (the same parameter is always redefined inside a block_begin - block_end body). Also
record types whose fields are declared as parameters are invalid. These restrictions are imposed by the structure of the symbol table since sometimes redundant information is stored for semantic analysis purposes and ignoring the previously mentioned lists contributes to getting rid of this redundancy. Clause mk_dic/1 returns a dictionary and is structured as

\[ mk_{-}\text{dic}(D) :- \text{read}(X), \]
\[ \begin{align*}
& ((X = [\ldots, \text{type}, \text{record}, [[.., [\ldots, \text{procedure}, \ldots]]]]) , \text{mk_{-}dic}(D)); \\
& ((X = [\ldots, [\ldots, \text{function}, \ldots]]), X = \ldots, [\ldots, \text{module\_header}, \ldots]); \\
& \text{app\_par}(X, [\text{Name}\_\text{Value}]), \text{lookup2}(\text{Name}, D, \text{Value}), \text{mk_{-}dic}(D)); \\
& ((X = \ldots, \text{block\_end}, X = \ldots, \text{block\_begin}, X = \ldots, [\ldots, \text{parameter}, \ldots]]), \text{mk_{-}dic}(D)); \\
& X = \ldots, \text{end\_of\_file}; \\
& (X = [\text{Name}\_\text{Value}], \text{lookup2}(\text{Name}, D, \text{Value}), \text{mk_{-}dic}(D))).
\end{align*} \]

Clauses for lookup2 are used in order to insert new entries into the dictionary:

\[ \begin{align*}
& \text{lookup2}(\text{Name}, \text{dic}(\text{Name}, \text{Value}, \ldots, \ldots), \text{Value}) :- !. \\
& \text{lookup2}(\text{Name}, \text{dic}(\text{Name1}, \ldots, \text{Before}, \ldots, \text{Value})), \text{Name} \text{ \#<\# Name1}, \text{lookup2}(\text{Name}, \text{Before}, \text{Value}). \\
& \text{lookup2}(\text{Name}, \text{dic}(\text{Name1}, \ldots, \text{After}, \ldots, \text{Value})), \text{Name} \text{ \#>\# Name1, \text{lookup2}(\text{Name}, \text{After}, \text{Value}).} \\
& \text{lookup2}(\text{Name}, \text{dic}(\text{Name}, \text{Value1}, \ldots, \ldots), \text{Value2}) :- \\
& \quad \text{write('---}), \text{nl}, \\
& \quad \text{write('Rename '}), \text{write(Name)}, \text{write(' ', ')}, \text{write(Value2)}, \\
& \quad \text{write('" if needed in the dictionary' \}), \text{nl}, \\
& \quad \text{write('Conflict with '}), \text{write(Name)}, \text{write(' ', ')}, \\
& \quad \text{write(Value1)}, \text{write('"' \)), \text{nl}, !.}
\end{align*} \]

The first clause inserts the new entry. The second and third clauses are responsible for finding where the new definition must be entered. If an attempt is made to enter a different value for an already existing name, the last clause notifies the user. The new definition is ignored. These clauses were produced by slightly modifying the lookup clauses from [25].

3.1.4. Performance of the Compiler Module

The performance of the compiler module was measured under an average system load of 1.18 Erlangs. This load corresponds to the number of jobs waiting in the CPU's queue. Each one of the protocol specifications used was compiled
ten times. The outcomes of each compilation were averaged and the results are
given in table 3.1. The size of the output is given in the column indicating the
size of the output syntax tree and it represents the syntax tree to be processed by
the rest of the modules. The size of the programs implementing the compiler
module and building the dictionary and syntax tree is 12830 lines.

<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input (lines)</th>
<th>size of output syntax tree (bytes)</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>242</td>
<td>1,988</td>
<td>2.57</td>
</tr>
<tr>
<td>simple transport</td>
<td>786</td>
<td>19,051</td>
<td>5.43</td>
</tr>
<tr>
<td>FTAM</td>
<td>1,750</td>
<td>41,904</td>
<td>11.93</td>
</tr>
<tr>
<td>LAP-D</td>
<td>2,769</td>
<td>73,356</td>
<td>16.35</td>
</tr>
</tbody>
</table>

Table 3.1. Performance of the compiler module
3.2. NORMALIZATION

The normalization module is responsible for a number of transformations which are applied to the initial specification. The objective of this module is to

a. Identify all possible paths of each transition,

b. define a predicate for each path (i.e. determine a boolean expression which should evaluate to true in order for the path to be executed) and

c. replace all variables -- including output parameters -- of each path with their symbolic values, [5].

Therefore, all statements resulting in transfer of control to a statement other than their next one must be eliminated. It is assumed that values of loop bounds can be determined statically. If this is not possible a limited number of iterations of the loop body is considered. This results in a non equivalent specification, but iterating over the loop body for three values of the loop conditional (i.e. two boundary values and one producing no iteration) and making sure that each variable changes at least once provides adequate testing, especially for while loops, [7]. This rule is not applied in the case of while loops only. In the case of for and all loops an error message is printed if the boundaries of the index variable cannot be statically defined. It is also assumed that procedures and functions are not defined recursively. The result of normalization is an Estelle specification which contains only single path transitions. The predicate for each path is moved to the provided clause.

The transformations applied are:

a. **With** statements replacement.

b. **Procedure** and **function** calls replacement.

c. Conditional (if and case) and repetitive (for, all, and while) statements replacement.
d. Elimination of major state sets from from clauses and replacement of element same in the to clause with the actual state.

The aforementioned transformations involve some modifications in the declaration part of the specification. Therefore, a processing of the declaration part is needed. In the sequel, the processing of declaration part and the details of each transformation will be discussed. Array data types and variant records defining PDUs are treated by the data flow analysis module. Finally, implementation of the normalization module will be explained.

3.2.1. Declaration Part Processing

Two types of Estelle declarations are processed initially:

a. procedures and functions and

b. variant records.

Procedure and function declarations are removed unless they are declared as primitive. The formal parameters of a procedure/function are considered first. If there exist value parameters, i.e. parameters preceded by the Estelle reserved word val, the possibility of their redefinition in the procedure/function body is examined. In this case a new global variable is created by appending the procedure/function name to the parameter name. All the occurrences of this value parameter in the procedure/function body are replaced by the newly created global variable. Then an assignment statement is added in the beginning of the begin-end block. This statement assigns the value parameter to its global substitute. Var parameters remain unchanged. Variable declarations local to a procedure/function are converted to global by appending the procedure/function name to the local variable name. Consequently, the local variable names are replaced by the global ones inside the procedure/function body. When the first procedure/function declaration is encountered a new dictionary is created.
Thereafter a dictionary entry is made for each procedure/function declaration. This entry contains the procedure/function name (and the function type, in case of functions), the list of formal parameters and the statement sequence consisting of the procedure/function body. This preprocessing facilitates the procedure/function replacement because it speeds up information retrieval.

A record definition containing a variant part is linearized unless it defines a PDU type and data flow analysis has been requested by the user. The term linearization means that the case construct is removed and the tag-field becomes a regular record field. If data flow analysis is required, records defining PDUs remain unchanged. This is necessary since the data flow analysis module will try to identify the PDUs and the fields belonging to each one of them. Consider, for example, the following variant record definition and assume that the user did not request data flow analysis:

\[
\text{T\text{PDUandCtrlInf}} = \text{record}
\]

\[
\begin{align*}
\{ & \text{control information} \} \\
\text{full} & : \text{boolean}; \\
\text{order} & : \text{orderTp}; \\
\text{peerAddr} & : \text{TAddrTp}; \\
\{ & \text{fields of T\text{PDU}} \} \\
\text{cr\text{Vl}} & : \text{creditTp}; \\
\text{destRef} & : \text{refTp}; \quad \{ \text{used for CR, CC, AK} \} \\
\text{srcRef} & : \text{refTp}; \quad \{ \text{used for CR, CC, DR, DC} \} \\
\text{user_data} & : \{ \text{optional} \} \text{dataTp}; \quad \{ \text{see TSAP; used for CR, CC, DR (not in this version of the protocol), DT, EDT} \} \\
\text{case kind} & : \text{TPDUCodTp of} \\
\text{CR}, \\
\text{CC} & : \{ \text{opts\_ind} \} : \text{OptTp}; \quad \{ \text{see TSAP} \} \\
\text{TSAPid\_calling}, \\
\text{TSAPid\_called} & : \text{T\_sufTp}; \quad \{ \text{optional} \} \\
\text{DR} & : \{ \text{is\_last\_PDU} \} : \text{boolean}; \quad \{ \text{control information} \} \\
\text{disc\_reason} & : \text{reasonTp}; \\
\text{DC} & : () ; \\
\text{DT} & : \{ \text{sendSeq} \} : \text{seqNumTp}; \\
\text{end\_of\_TSDU} & : \text{boolean}; \\
\text{AK} & : \{ \text{exp\_sndSeq} \} : \text{seqNumTp}; \\
\text{undef\_code} & : () ; \quad \{ \text{end of case} \}
\end{align*}
\]

end; \{ \text{of T\text{PDUandCtrlInf}} \}
The result of linearization is

\[ tpduandctrlinf =
\begin{align*}
&\text{record} \\
&\quad \text{full: boolean;} \\
&\quad \text{order: ordertype;} \\
&\quad \text{peeraddr: taddrtp;} \\
&\quad \text{crul: seqnumtp;} \\
&\quad \text{destref: reftp;} \\
&\quad \text{srcref: reftp;} \\
&\quad \text{user_data: datatp;} \\
&\quad \text{kind: tpduodtp;} \\
&\quad \text{opts_ind: opttp;} \\
&\quad \text{tsapid_calling: t_suftp;} \\
&\quad \text{tsapid_called: t_suftp;} \\
&\quad \text{is_last_pdu: boolean;} \\
&\quad \text{disc_reason: reasontp;} \\
&\quad \text{sendseq: seqnumtp;} \\
&\quad \text{end_of_tsdu: boolean;} \\
&\quad \text{expseq: seqnumtp}
\end{align*}
\]

New variables are created during the production of normal form transitions, as we shall see later in this chapter. Information concerning these variables is stored in a list structure during transitions processing. This structure is used in order to derive new declarations and add them to the declaration part of the normalized specification.

3.2.2. With Statements Replacement

The general structure of with statement is

\[ \text{with } v_1, v_2, \ldots, v_n \text{ do } s \]

where \( v_n, n = 1, 2, \ldots, \) are the variables in the record variable list of with statement. Each selector name inside statement \( s \) is prefixed with the record variable name which is referred to by this selector. Then the with statement is replaced by statement \( s \). For example the with statement block

\[ \text{with } PDU \text{ do} \\
\quad \text{begin} \\
\quad \quad \text{kind} := CR; \\
\quad \quad \text{peerAddr} := destAddr; \]

where selectors kind, peerAddr, Opts_ind, crVl and order refer to record variable PDU, is changed to

\[
pdu.kind := cr; \\
pdu.peeraddr := destaddr; \\
pdu.opts_ind := opts; \\
pdu.crvl := rcr; \\
pdu.order := first;
\]

### 3.2.3. Procedure and Function Calls Replacement

Each transition block is scanned in order to identify procedure or function calls. If a procedure or function call is found, a lookup in the dictionary is invoked in order to obtain the formal parameters and the statement sequence comprising the procedure or function body. The process of replacing a procedure call differs from the one for a function call.

#### 3.2.3.1. Procedure Calls

Procedure calls are treated in the following way. The procedure block is scanned and all the occurrences of formal parameters are replaced by their actual values (i.e. the variables or expressions given as arguments at the time of call). Then the modified procedure block replaces the procedure call. The transition

\[
\text{trans} \\
\text{from ESTAB} \\
\text{to ACK_WAIT} \\
\text{when U SEND_request} \\
\text{begin} \\
\quad \text{copy}(P.\text{Msgdata}, \text{Udata}); \\
\quad B.\text{Seq} := \text{Send_seq}; \\
\quad \text{Store}(\text{Send_buffer}, P); \\
\quad M\text{sgseq} := \text{Send_seq}; \\
\quad \text{Format_data}(P, B); \\
\quad \text{output N.DATA_request}(B); \\
\quad \text{end}; \\
\]

where Format_data is
procedure Format_data(Msg: Msg_type; var B: Ndata_type);
begin
  with B do
  begin
    Id := Dt;
    Conn := Conn_end_pt_id;
    copy(Data, Msg.Msgdata);
  end;
end;

is transformed to

trans
{1}
when u.send_request
from estab
to ack_wait
begin
  copy(p.msgdata, udata);
  b.seq := send_seq;
  store(send_buffer, p);
  msgseq := send_seq;
  b.id := dt;
  b.conn := conn_end_pt_id;
  copy(b.data, p.msgdata);
  output n.data_request(b)
end;

3.2.3.2. Function Calls

In the case of a function call, a global variable is created from the function
name by appending a unique number at the end of the function name. This vari-
able is added to the declaration part after the processing of the transitions
finishes. The function body is retrieved from the dictionary and the new name
substitutes the function name inside its body. The modified function body is
inserted just before the function call and the function call is replaced by the
newly derived global variable. Consider the call of function DR_PDU

WHEN Map.transfer { PDU }
PROVIDED (PDU.kind = CC) and not (PDU.Opts_ind <= Opts)
FROM waitCC
TO waitDC
begin
  OUTPUT TS.TDISind(protocol_error);
  OUTPUT Map.transfer(DR_PDU(protocol_error,true));
end;

where $DR\_PDU$ is defined as

```pascal
function DR_PDU(r: reasonTp; last_PDU: boolean): TPDUandCtrlInf;
var PDU : TPDUandCtrlInf;
begin
  with PDU do
  begin
    kind := DR;
    disc_reason := r;
    is_last_PDU := last_PDU;
    order := destructive;
  end;
  DR_PDU := PDU;
end;
```

Then normalization produces

```pascal
trans
{ 00 }
when map.transfer
provided (pdu.kind = cc) and not (pdu.opts_ind <= opts)
from waitce
to waitdc
begin
  output ts.tdisind(protocol_error);
  pdu_dr_pdu.kind := dr;
  pdu_dr_pdu.disc_reason := protocol_error;
  pdu_dr_pdu.is_last_pdu := true;
  pdu_dr_pdu.order := destructive;
  dr_pdu_9 := pdu_dr_pdu;
  output map.transfer(dr_pdu_9)
end;
```

A problem may arise when a function is called within a `provided` clause. If

- the function body consists of a single statement assigning an expression to
  the function identifier or

- the function body can be reduced by symbolic execution to a single state-
  ment assigning to the function identifier a symbolic expression

then this expression replaces the function call. It is implied that the formal
parameters will be substituted by the actuals when the function replacement is
performed. As an example to the first case the transition

```pascal
trans
from ACK_WAIT
to ESTAB
  when N.DATA_response
  provided Ack_ok(Ndata)
  begin
    Remove(Send_buffer);
    Inc_send_seq;
  end;

where Ack_ok is declared as

function Ack_ok(Nd: Ndata_type): boolean;
begin
  Ack_ok := (ND.Id = ACK) and (Nd.Seq = Send_seq);
end;

is transformed to

trans
  { 5 }
  when n.data_response
  provided (ndata.id = ack) and (ndata.seq = send_seq)
  from ack_wait
  to estab
  begin
    remove(send_buffer);
    send_seq := (send_seq + 1) mod 2
  end;

Next we give an example to the second case. If the function

function OrderConstraint(T_suf: T_sufTp;
  EPId : TCEPIdTp;
  kind : TPDUCodTp) : boolean;

var OK : boolean;
begin
  OK := true;
  with TC[T_suf,EPId] do
    ALL k : TPDUCodTp do
      if (k <> kind) and PDU_buf[the].full
      and (PDU_buf[k].order < PDU_buf[kind].order)
      then OK := false;
    OrderConstraint := OK;
end;

is called, symbolic execution results in the equivalent single statement function †

function orderconstraint(t_suf: t_sutfp; epid: tcepidtp; kind: tpducodtp)

† This is a special case where it is possible to produce a symbolic value because the order of iterations of all's body is not significant and the value of variable OK is just the logical inversion of if's boolean expression.
begin
  orderconstraint := not(
    (cr <> kind) and
    pdu_buf[cr].full and
    (pdu_buf[cr].order < pdu_buf[kind].order)
  or
    (cc <> kind) and
    pdu_buf[cc].full and
    (pdu_buf[cc].order < pdu_buf[kind].order)
  or
    (dr <> kind) and
    pdu_buf[dr].full and
    (pdu_buf[dr].order < pdu_buf[kind].order)
  or
    (dc <> kind) and
    pdu_buf[dc].full and
    (pdu_buf[dc].order < pdu_buf[kind].order)
  or
    (dt <> kind) and
    pdu_buf[dt].full and
    (pdu_buf[dt].order < pdu_buf[kind].order)
  or
    (ak <> kind) and
    pdu_buf[ak].full and
    (pdu_buf[ak].order < pdu_buf[kind].order)
  or
    (ndef_code <> kind) and
    pdu_buf[ndef_code].full and
    (pdu_buf[ndef_code].order < pdu_buf[kind].order)
  )
end;

If a call to the function OrderConstraint occurs in the boolean expression of a
provided clause, the left hand side expression assigned to variable OK replaces
the function call.

If the function body contains more than one statement and symbolic execution
fails to determine an expression for the function identifier (e.g. when a
primitive procedure call occurs), no transformation is attempted and the user
is informed with message

line L: Function_Id is not a single statement function

where L is the number of the line containing the function declaration and
Function_Id is the name of the function. The reason is that replacement of a function call with function's body is not possible if the call occurs inside a provided clause. Estelle does not provide any mechanism which enables insertion of a statement sequence just before provided's condition is evaluated. Any statements implied by provided's condition are executed internally by the process implementing the module and no interaction is assumed.

3.2.4. Conditional Statement's Replacement

If statements inside a transition block are removed by creating new transitions for each logical value (true or false) of the condition. This logical value of each condition defines a path inside the transition body. Therefore, a path is valid if a certain predicate evaluates true. This predicate is moved to the provided clause and the statements associated with the corresponding path comprise the transition body. If a variable occurring inside the condition of an if statement is assigned a value in the preceding statements a symbolic replacement is applied: the symbolic value of the variable is computed by symbolic execution and this value replaces the variable in the boolean expression of the if statement.

Consider the transition

TRANS
WHEN Map.transfer { PDU }
PROVIDED PDU.kind = AK
FROM open
TO SAME
var newCr : 0 .. 255;
begn
with PDU do
  begin
    newCr := crVl + expSndSeq - TSseq;
    if newCr >= SCr then
      SCr := newCr
    else error(newCr);
  end;
end;

This transition produces two transitions (also notice the expansion of the with
statement)

    trans
    { 024 }
    when map.transfer
    provided (pdu.kind = ak) and
       (not (pdu.crul + pdu.expseq - tsseq >= scr))
    from open
    to open
    begin
       error(pdu.crul + pdu.expseq - tsseq)
    end;

    trans
    { 025 }
    when map.transfer
    provided (pdu.kind = ak) and
       (pdu.crul + pdu.expseq - tsseq >= scr)
    from open
    to open
    begin
       scr := pdu.crul + pdu.expseq - tsseq
    end;

The user is notified for a symbolic replacement with message

    line L: Var_Id replaced by its value,

where L is the line number and Var_Id is the variable identifier.

Case statements are replaced in a similar way. For each case constant a
new predicate is created. This predicate is an equality consisting of the case-
index (the expression in the header of the case statement) on the left-hand side
and the case constant on the right-hand side. Since the index-expression should
evaluate to one of the case constants, the paths corresponding to the case state-
ment are explicitly defined (if the case-index takes a value other than the ones
specified a run-time error occurs). A new transition is created for each predicate
and its corresponding path.

3.2.5. For, All and While Statements Replacement

If the index variable of a loop statement (like for, all and while) has stati-
cally defined values, the statement body is repeated for each one of these values.

Consider for example the following for loop

```
INITIALIZE
TO Idle
var kind: TPDUCodTp;
beg
   ALL T_suf : T_sufTp DO
      ALL EPLd : TCEPIDTp DO
      with TC[T_suf, EPld] do
         begin
            ClsClnBfrs(T_suf, EPld);
            for kind := CR to AK do
               PDU_buf[kind].is_last_PDU := false;
               PDU_buf[DC].is_last_PDU := true;
         end;
      end;
end;
```

Since variable kind is of type TPDUCodTp it can take one of the values CR, CC, DR, DC, DT, AK, undef_code. Procedure ClsClnBfrs is defined as

```
procedure ClsClnBfrs(T_suf : T_sufTp; EPld : TCEPIDTp);
var
   kind : TPDUCodTp;
   undef : NCEPIDTp;
beg
   with TC[T_suf,EPld] do
      begin
         asgned_NC := undef;
         for kind := CR to AK do
            PDU_buf[kind].full := false;
      end;
end;
```

Then the initialize clause is transformed to

```
initialize
to idle
begin
   all t_suf: t_sufTp do
      begin
         all epid: tcepidTp do
            begin
               tc[t_suf, epid].asgned_NC := undef_clsClnBfrs;
               tc[t_suf, epid].pdu_buffer[cr].full := false;
               tc[t_suf, epid].pdu_buffer[cc].full := false;
               tc[t_suf, epid].pdu_buffer[dr].full := false;
               tc[t_suf, epid].pdu_buffer[dc].full := false;
               tc[t_suf, epid].pdu_buffer[dt].full := false;
               tc[t_suf, epid].pdu_buffer[ak].full := false;
               tc[t_suf, epid].pdu_buffer[cr].is_last_pdu := false,
```
Similarly, all statement

\[ \text{ALL } k : \text{TTPUCodTp DO } \text{T}|T \_suf, \text{EPIId}|.\text{PDU}_{-}buf[k].\text{full} = \text{false}; \]

is expanded to

\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buffer}[k].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[cc].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[dr].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[dc].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[dt].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[ak].\text{full} = \text{false}; \]
\[ \text{tc}|T \_suf, \text{epid}|.\text{pdu}\_buf[undef\_code].\text{full} = \text{false} \]

In the above examples the index variable was statically defined, therefore exhaustive enumeration was possible. If this is not possible (i.e. when the index values change dynamically) then, in the case of while statements, a limited number of iterations is assumed (in this system we consider only three iterations). In the case of for statements a warning of the form

\[ \text{line } L : \text{failed to determine range values of Index in FOR statement}, \]

where \( L \) is the line number and Index is the index variable, is displayed. The while loop is treated as follows: For each of the values that make the guard expression true a different path is created. Another path is created for a value of a variable that makes the guard expression false (i.e. this path does not include execution of the loop body). If the boolean expression contains dynamically changing variables three paths are taken: one with the expression evaluating to false and two with values chosen so that the loop body will be repeated once for the second path and twice for the third. Each path, as mentioned before,
corresponds to a new transition. For example, the transition:

TRANS
ANY T_suf : T_sufTp;
   EPIId : TCEPIIdTp;
   NCId : NCEPIIdTp DO
PROVIDED TC[T_suf].EPIId.[PDU_buf[CR]].full and
   (TC[T_suf].EPIId.[PDU_buf[CR]].peerAddr.N_pref ==
   NC[NCId].[remoteNaddr])
begin
   with TC[T_suf, EPIId], NC[NCId] do
   begin
      assgnd_NC := NCId;
      ref := 1;
      while ref in activeRefs do ref := ref + 1;
   end;
end;

is expanded to

trans
any t_suf: t_suftp; epid: tcepidtp; ncid: ncepidtp do
provided (tc[t_suf, epid].pdu_buf[cr].full and
   (tc[t_suf, epid].pdu_buf[cr].peeraddr.n_pref == nc[ncid].remotenaddr))
from idle
to idle
begin
   tc[t_suf, epid].assgnd_nc := ncid;
   ref_ascgnnewref := 1;
   ref_ascgnnewref := ref_ascgnnewref + 1;
   ref_ascgnnewref := ref_ascgnnewref + 1;
end;

trans
any t_suf: t_suftp; epid: tcepidtp; ncid: ncepidtp do
provided (tc[t_suf, epid].pdu_buf[cr].full and
   (tc[t_suf, epid].pdu_buf[cr].peeraddr.n_pref == nc[ncid].remotenaddr))
from idle
to idle
begin
   tc[t_suf, epid].assgnd_nc := ncid,
   ref_ascgnnewref := 1;
   ref_ascgnnewref := ref_ascgnnewref + 1;
end;

trans
any t_suf: t_suftp; epid: tcepidtp; ncid: ncepidtp do
provided (tc[t_suf, epid].pdu_buf[cr].full and
   (tc[t_suf, epid].pdu_buf[cr].peeraddr.n_pref == nc[ncid].remotenaddr))
from idle
to idle
begin
   tc[t_suf, epid].assgnd_nc := ncid;

ref_assignnewref := 1;
end;

where activeRefs has been declared as a set of integers and

ref := 1 for activeRefs = { }
ref := 2 for activeRefs = { 1 }
ref := 3 for activeRefs = { 1, 2 }.

3.2.6. Processing of From and To Clauses

Finally, major state lists or sets in the from clause are eliminated by repeating the transition for each state value in the state list. For example a state set of the form

\[
\text{any-state} = \{ \text{closed, waitCC, waitTCONresp, open, waitDC, closing} \};
\]

in a from clause will cause the generation of six normal form transitions.

The element same in the to clause is replaced by the actual destination state (i.e. the state in the from clause). Therefore the transition

\[
\text{trans} \\
\text{from EITHER} \\
\text{to same} \\
\text{when U.RECEIVE_request} \\
\text{provided not buffer_empty(Recv_buffer)} \\
\text{begin} \\
\text{Q.Msgdata := Retrieve(Recv_buffer);} \\
\text{output U.RECEIVE_response(Q.Msgdata);} \\
\text{Remove(Recv_buffer)} \\
\end;
\]

where EITHER = [ACK_WAIT, ESTAB] is a state set, is replaced by two trans-

sitions:

\[
\text{trans} \\
\text{\{ 2 \} } \\
\text{when u.receive_request;} \\
\text{provided not buffer_empty(recv_buffer)} \\
\text{from estab} \\
\text{to estab} \\
\text{begin}
\]
\[ \ldots \{ \text{same as above} \} \]
end;

trans
{ S }
when u.receive_request
provided not buffer_empty(recv_buffer)
from ack_wait
to ack_wait
begin
\[ \ldots \{ \text{same as above} \} \]
end;

3.3. IMPLEMENTATION OF NORMALIZATION MODULE

The compilation phase generates the syntax tree of the input specification. During normalization this tree is changed. Nodes are deleted or new nodes are created. Initially, the syntax tree is read and submitted to the normalization routines. Then it is scanned top-down in order to process the declaration, initialization and transition declaration part. Also a global dictionary is created.

Normalization is done for each module body separately. Clauses \( n/2 \) implement the normalization algorithm and are defined recursively. A different clause for each subtree representing an Estelle construct (e.g. a module body definition) is defined. Initially, the complete syntax tree is unified with the first argument of \( n/2 \). Then \( n/2 \) is called again with its first argument instantiated to one of the subtrees hanging off the root and so on. For example, the Prolog routine

\[
\begin{align*}
n(mdBdD/ & \quad \text{id}(N,L), \\
& \quad \text{Id}, \\
& \quad bdD/( \\
& \quad \quad \text{DclPrt0, InitPrt0, TrDclPrt0} \\
& \quad ), \\
mbD/ & \quad \text{id}(N,L), \\
& \quad \text{Id}, \\
& \quad bdD/( \quad \\
& \quad \quad \text{UclPrt, InitPrt, TrDclPrt} \\
& \quad ) \\
\end{align*}
\]
calls the routines to normalize the subtrees hanging off the current module definition. Unification instantiates variables DclPrt0, InitPrt0 and TrDclPrt0 with the subtrees representing the original structure of declaration, initialization and transition declaration part, respectively. Notice the recursive nature of this clause: the declaration part, for example, has a subtree corresponding to a declaration and a second subtree corresponding to a declaration part (because of the recursive grammar used for the implementation of this construct). One of the sub-declarations could be a module body definition. Then, the same clause will handle the new declaration. The result will be a tree of the normalized construct.

Since Prolog does not permit global variables or flags, clause assertion has been used to store global information. This information mainly refers to new variables created by procedure or function replacement and major states. The set of all major states is needed for the from and to clauses processing.

Linearization of a variant record uses the dictionary entry of the variant record and produces a Prolog tree for the linear record definition:

\[
\text{linearize} (TpNm, FldLst) :- \\
\text{current_dic}(Dic), \\
\text{lookup}(TpNm, Dic, [{\text{type, \{record, Lst_of_fields\}}}], \\
\text{reverse}(Lst_of_fields, [], L), \\
\text{mkFldLst}(T, FldLst), !.
\]

Variable FldLst is instantiated when the clause exits to a Prolog tree representing the fields of the record type named after the value of variable TpNm (type
When the subtree of a **procedure** or **function** declaration is encountered (i.e. the Prolog subtree unifies with a Prolog structure defining a procedure or function declaration) the following actions are taken:

- local variables are changed to global (clauses `chng VrToGlob/4`) in the way explained in 3.2.1 and
- the procedure/function declaration along with its statement block is saved in a dictionary (clause `storeInTmp/2`) and removed from the syntax tree.

According to a restriction imposed by the NBS compiler, the procedure and function declaration part inside the scope of a procedure/function is assumed empty (i.e. it takes always the atomic value `prcAndFncDclPrt`). The implementation of the clauses for treating procedure or function declarations is straightforward:

```prolog
n(dclPrt/
  DclPrt0,
  dcls/
    prcDcl/
      prcHd(_, id(N, _), _),
      blk(bldclPrt, cnstDjPrt, tpDjPrt, VrDclPrt,
        prcAndFncDclPrt, stmtPrt(StmtSeq0))
    }
  }
  DclPrt) :-
    ((VrDclPrt = vrDclPrt(VrDcls),
      chng VrToGlb(VrDcls, N, StmtSeq0, StmtSeq)),
     StmtSeq = StmtSeq0),
    n(DclPrt0, DclPrt),
    /* Store procedure declaration and statement sequence in clause 'tmp_dic(X)' */
    storeInTmp(N, StmtSeq), !.

n(dclPrt/
  DclPrt0,
  dcls/
    fncDcl/
      fncHd(_, id(N1, _), _FFPL, _id),
      blk(bldclPrt, cnstDjPrt, tpDjPrt, VrDclPrt,
        prcAndFncDclPrt, stmtPrt(StmtSeq0))
    }
  })
```

Normalization clauses for the initialization and transition declaration part are implemented in the same way. We will examine how normalization of transitions is performed keeping in mind that the same things apply for the initialization construct (except that the initialization construct has only provided and to clauses). Normalization of transitions is done in three steps, executed sequentially. The routines needed for each step are called by the clause:

\[ n(\text{trGr(Cls0, TrBleck0, TrGrs}) : \]

\[ /* \text{STEP 1: Replace procedure and function calls and remove WITH, FOR, ALL and WHILE statements} */ \]

\[ n1(\text{trGr(Cls0, TrBleck0, TrGrsA}), !, \]

\[ /* \text{STEP 2: Remove conditional statements (IFs and CASEs}) */ \]

\[ n2(\text{TrGrsB, TrGrsC}), !, \]

\[ /* \text{STEP 3: Process FROM and TO clauses} */ \]

\[ n3(\text{TrGrsC, TrGrs}), !. \]

The first step is represented by clause \text{n1/2} and calls \text{lookForProcOrFuncCall/2, repWithStmts/2, repWhile/3, and repForStmts/2} in order to replace

a. procedure and function calls,

b. with statements,

c. while statements,

d. for and all statements,

respectively. Since while statements introduce paths in the transition block, the output of the first step is a number of transitions represented by tree \text{TrGrsA} (transition groups A).
Clauses `lookForProcOrFuncCall` scan the statement sequence of the transition block recursively (each statement sequence tree consists of two subtrees: one for a statement sequence -- here comes the recursion -- and one for a statement).

A `procedure` call can occur only as an identifier followed by an expression list (possibly empty) containing the actual parameters:

```
lookForProcOrFuncCall(stmt(id(N, _), XprLst), StmtSeq) :-
  lookup_tmp(N, [[procedure, []], Prms, ProcStmts]),
  /* Replace procedure formal(s) by actual(s) */
  repPrms(N, Prms, XprLst, ProcStmts, StmtSeqA),
  lookForProcOrFuncCall(StmtSeqA, StmtSeq), !.
lookForProcOrFuncCall(stmt(id(N, _)), StmtSeq) :-
  lookup_tmp(N, [[procedure, []], _params, ProcStmts]),
  lookForProcOrFuncCall(ProcStmts, StmtSeq), !.
```

A function call can occur in an expression or simply as a variable access. A new name for the function's identifier is created each time a function call replacement is done. This name and its type are asserted in Prolog clause `vrDcl` (variable declaration) in order to be appended later to the declaration part of the normalized specification:

```
lookForFuncCall(xpr(id(N0, L), XprLst), Xpr, StmtSeq) :-
  lookup_tmp(N0, [[function, [Tp]_]], Prms, ProcStmts),
  mkNewNm(N0, N, [Tp]),
  Xpr = xpr(vrAcc(id(N, L))),
  retract(vrDcl(Vdl)), append([[N, Tp]], Vdl, NewVdl),
  assert(vrDcl(NewVdl)), !,
  repNm(N0, N, ProcStmts, StmtSeqA),
  /* Replace function formal(s) by actual(s) */
  repPrms(N, Prms, XprLst, StmtSeqA, StmtSeqB),
  lookForProcOrFuncCall(StmtSeqB, StmtSeq), !.
lookForFuncCall(vrAcc(id(N0, L)),
    vrAcc(id(N, L)), StmtSeq) :-
  lookup_tmp(N0, [[function, [Tp]_]], _params, ProcStmts),
  mkNewNm(N0, N, [Tp]),
  retract(vrDcl(Vdl)), append([[N, Tp]], Vdl, NewVdl),
  assert(vrDcl(NewVdl)), !,
  repNm(N0, N, ProcStmts, StmtSeq1),
  lookForProcOrFuncCall(StmtSeq1, StmtSeq), !.
```

Notice that after a procedure/function call replacement the statement sequence produced is checked for other procedure/function calls.
The heart of symbolic replacement of identifiers are clauses \texttt{repNm/4} (replace name). Clauses \texttt{repNm/4} are called by any clause attempting symbolic replacement (e.g. \texttt{repPrms/6} -- replace parameters). These clauses search a Prolog tree in order to find all references to an identifier. When a reference is found the symbolic replacement is done according to rules specified as \texttt{repNm/4} clauses. For example, the rule for replacement of an identifier by an expression is
\[
\texttt{repNm}(N_1, N_2, xpr(vrAcc(id(N_1, _))), N_2) :-
\]
\[
N_2 =.. [xpr|_/], !.
\]
where identifier \(N_1\) must be replaced by expression \(N_2\). The predicate \(N_2 =.. \) \(\downarrow\) makes sure that \(N_2\) is an expression. Clauses \texttt{repNm} are based on Prolog’s predicate \texttt{univ (=..)} in order to isolate the children of a Prolog tree:
\[
\texttt{repNm}(N_1, N_2, S, NewS) :-
\]
\[
S =.. [H|T],
\]
\[
\texttt{repNm1}(N_1, N_2, T, NewT),
\]
\[
NewS =.. [H|NewT], !.
\]
\[
\texttt{repNm1}(_, _, [], []), !.
\]
\[
\texttt{repNm1}(N_1, N_2, [X], [NewX]),
\]
\[
\texttt{repNm}(N_1, N_2, X, NewX),
\]
\[
\texttt{repNm1}(N_1, N_2, Y, NewY), !.
\]
A recursive search is performed on each child until all the children are covered. The expression \(X =.. L\) means that \(L\) is the list consisting of the functor of \(X\) followed by the arguments of \(X\). Functor is the name of a structure and is written just before the structure's opening parenthesis.

Replacement of \texttt{with} statements is straightforward. This routine is also recursive in order to cover nested \texttt{withs}:
\[
\texttt{replWithStmts(stmt(withStmt(rcrdVrLst(VrAcc), Stmt)), StmtSeq)} :-
\]
\[
\texttt{appFldNm(VrAcc, Stmt, StmtSeq1)},
\]
\[
\texttt{replWithStmts(StmtSeq1, StmtSeq), !}.
\]
\[
\texttt{replWithStmts(stmt(withStmt(rcrdVrLst(RcrdVrLst, VrAcc), Stmt)), StmtSeq)} :-
\]
\[
\texttt{appFldNm(VrAcc, Stmt, StmtSeq1)},
\]
\[
\texttt{replWithStmts(StmtSeq1, StmtSeq1)},
\]
\[
\texttt{replWithStmts(stmt(withStmt(RcrdVrLst, stmt(empStmt(StmtSeq1)))), StmtSeq2)},
\]
\[
\texttt{replWithStmts(StmtSeq2, StmtSeq), !}.
\]

When a \texttt{while} statement is found its body is scanned for nested \texttt{while} state-
ments. Each nested while produces a number of paths, each path having a predicate. For each of those paths more paths are created because of the outer while and so on:

\[ \text{repWhile}(\text{Cls, stmtSeq}(\text{StmtSeq}, \text{Stmt}), \text{PathLst}) :- \]
\[ \quad \text{\{Stmt = stmt(cmpStmt(_));} \]
\[ \quad \quad \text{Stmt = stmt(while}_\text{do, _}, _)_\text{;}, !_; \]
\[ \quad \text{repWhile}(\text{Cls, stmtSeq, PathLst1}), \]
\[ \quad \text{addStmt(PathLst1, Stmt, PathLst)} \]

The for and all statements are treated by the same clauses because of the similarity in their structure. It is assumed that the index variables are of a type with statically defined values (this is mandatory for the all statement). After clauses getPrmLst/5 get the values of the index variable in Prolog list Lst, the for’s or all’s block is repeated for each value. The produced sequence of statements is also scanned for for/all statements:

\[ \text{repForStmts(stmt(for}_\text{to}_\text{do, id(N, L), Xpr1, Xpr2, Stmt), StmtSeq)} :- \]
\[ \text{getPrmLst(for}_\text{to, id(N, L), Xpr1, Xpr2, Lst}, \]
\[ \text{iterate(N, Lst, stmtSeq(Stmt), StmtSeq1)}, \]
\[ \text{repForStmts(StmtSeq1, StmtSeq), !.} \]

\[ \text{repForStmts(stmt(allStmt(domLst(dom(idLst(id(N0,L))), smplTp(id(T,_,))))),} \]
\[ \text{Stmt0)),} \]
\[ \text{StmtSeq)} :- \]
\[ \text{\{find(T, [[type, [scalar, [Low|Rest]]]],} \]
\[ \text{last([Low|Rest], High)}; \]
\[ \text{mkNewNm(N0, N, [T])}, \]
\[ \text{repNm(N0, N, Stmt0, Stmt)}, \]
\[ \text{repForStmts(stmt(for}_\text{to}_\text{do,} \]
\[ \text{id(N, L),} \]
\[ \text{xpr(veAcc(id(Low,0))),} \]
\[ \text{xpr(veAcc(id(High,0))),} \]
\[ \text{Stmt)}, \]
\[ \text{StmtSeq), !.} \]

Clauses n2/2 implement the second step. The basic clauses called by n2/2 are:

- getPath/3 finds all possible paths in the statement sequence of the transition block and creates the Prolog list pathLst whose elements are the paths and their predicates. getPath/3 also check if some variables need symbolic
replacement. If yes the clauses that do the checking assert the clause `symRepList/1` which contains the information about the variables to be replaced.

- `mkTr/3` makes one transition for each path. It is implemented using tail recursion on the path list. The result is a tree with transition groups, each group consisting of one transition.

- If there are variables needing to be symbolically replaced (i.e. clause `symRepList/1` exists) the transition groups produced by `mkTr/3` are modified accordingly. `getSymVals/3` scans each transition in order to get the symbolic value of variables in the list argument of `symRepList/1` and replace it in the provided clause.

The main clause of the `n2/3` clauses is

```prolog
n2(trGr(Cls0, 
    trBick(CDP, TDP, VDP, PAFDP, TN, cmpStmt(StmtSeq0)), 
    TrGrS), :- 
    /* Get each path in 'StmtSeq0' */ 
    getPaths(Cls0,StmtSeq0,PathLst), 
    /* Make one transition for each path */ 
    mkTr(PathLst, [CDP, TDP, VDP, PAFDP, TN|, TrGrS1], 
        (retract(symRepList(L)), 
        getSymVals(TrGrS1, TrGrS, L)); 
    TrGrS = TrGrS1), !.
```

Consider `getPaths/3` for if statements as an example. Each path is checked recursively for other paths introduced by nested conditionals:

```prolog
getPaths(Cls0, 
    stmtSeq(StmtSeq, stmt(if_then_else, bXpr(Xpr), Stmt1, Stmt2)), 
    PathLst) :- 
    checkSymb(StmtSeq, Xpr, [], _NmLst), 
    getPaths(Cls0, stmtSeq(StmtSeq, Stmt1), PathLstA), 
    addPred(PathLstA, Xpr, PathLstB), 
    getPaths(Cls0, stmtSeq(StmtSeq, Stmt2), PathLstC), 
    addPred(PathLstC, xpr(not, xpr(Xpr)), PathLstD), 
    append(PathLstB, PathLstD, PathLst), !.
```

Clauses `mkTr/3` build one transition for the head of each path list given as an argument. The same clauses are called with the tail of the list and so on until all the list elements are covered:
\[ mkTr(\]
\[ [[Cls, Stmt_seq]],\]
\[ [CDP, TDP, VDP, PAFDP, TN],\]
\[ trGr(\]
\[ trCr(\]
\[ Cls,\]
\[ trBlck(CDP, TDP, VDP, PAFDP, TN, cmpStmt(Stmt_seq))\]
\[ )\]
\) :- !. \[ mkTr([Path|RestPaths], Dcls, TrGr) :-\]
\[ mkTr([Path], Dcls, TrGr1),\]
\[ mkTr(RestPaths, Dcls, TrGr2),\]
\[ appendTrGrs(TrGr2, TrGr1, TrGr).\]

Clauses \textit{getSymVals/3} and \textit{getSymVals/4} are based on recursively searching the tree of transition groups in order to identify the statement sequence that produces the symbolic value that replaces a variable in the \textit{provided} clause:

\[ getSymVals(ccls(Cls, cl(provCl(blXpr(Xpr0)))),\]
\[ ccls(Cls, cl(provCl(blXpr(Xpr))));\]
\[ StmtSeq,\]
\[ L), :-\]
\[ symbRep(StmtSeq, \text{ } Xpr0, Xpr, L), !.\]

Finally, the third step is performed by \textit{proFromAndTo/2}. Its implementation is rather direct according to what is described in section 3.2.6.

\[ proFromAndTo(trGr(Cls0, TrBlck), TrGr) :-\]
\[ / * Process TO clause */\]
\[ proTo(Cls0, ClsA, State),\]
\[ / * If TO clause is omitted next state is 'same' */\]
\[ (State = same; true),\]
\[ / * Process FROM clause */\]
\[ proFrom(ClsA, ClsB, StateList),\]
\[ / * If FROM clause is omitted the transition applies to all states in the specification */\]
\[ ((\text{var(StateList)}, \text{all_states(StateList)}); \text{nonvar(StateList)}),\]
\[ popWhn(ClsB, ClsC),\]
\[ popAny(ClsC, ClsD),\]
\[ genTr(ClsD, TrBlck, StateList, State, TrGr), !.\]

Routines \textit{popWhn} and \textit{popAny} move the Estelle clauses \textit{when} and \textit{any} to the beginning of the transition clauses \textit{any} comes first). Thus it is made sure that transition clauses appear in a certain order in the normalized specification (\textit{any, when, provided/delay/priority, from, to}).
Future modifications of the normalization module can be made easily because of the small size of Prolog clauses and the declarative and modular programming style followed. It has become clear that each Estelle construct is mapped to an equivalent Prolog tree fragment which undergoes all the processing. This tree fragment has a specific structure imposed by the grammar rules of Estelle. Modifications to the rules describing an Estelle construct imply modifications to the related heads of clauses or data structures. If the programmer understands the tree structure resulting from the Estelle compiler rules, the changes or additions of clauses become trivial. Utility routines that apply on many different constructs have been designed as general as possible. For example clauses *repN*m/*4*, discussed earlier in this section, require minimum change because of the use of *unit* operator. Even though *unit* imposes a non-declarative programming style and makes the program cryptic, it has been used for fundamental actions such as symbolic replacement of identifiers, parameters, procedures and functions.

### 3.3.1. Performance of Normalization Module

The performance was measured under an average CPU load of 1.28 Erlangs. The execution times presented in table 3.2 are the averages of the time results of ten experiments for each protocol. Columns 'size of input syntax tree' and 'size of output syntax tree' give the size of data processed and produced, respectively. The number of transitions give a rough idea of the increase of the specification size. Notice that the largest specification (LAP-D protocol) is normalized faster than FTAM due to the lower complexity of path producing constructs in LAP-D. FTAM contains more declarations that have to be processed and expanded. These declarations are usually variant records which are linearized and procedure or function calls which should be placed in the dictionary for reference during
normalization. LAP-D specification contains only one variant record and most of the procedures or functions are declared as primitive. The normalization module required 1496 lines of Prolog code for its implementation.

3.3.2. Printing

The printing clauses print an Estelle specification from its parse tree. The tree is depth-first searched starting from the root when a call to the routine

\[
p(\text{spec(} \text{Id, SstmClass, DfltOpt, TmOpt, BdDfl, I}) \text{):-}
\]

\[
\text{write(}'\text{specification}'\text{), } p(\text{Id}), \quad / \quad \text{: specification header */}
\]

\[
p(\text{SstmClass, I}), \quad \text{\textit{write(}'\text{,}'\text{), */ system class */}}
\]

\[
p(\text{DfltOpt, I}), \quad / \quad \text{\textit{write(}'\text{ },'\text{), */ default options */}}
\]

\[
p(\text{TmOpt, I}), \quad / \quad \text{\textit{write(}'\text{ },'\text{), */ time options */}}
\]

\[
\text{NewI is I + 4,} \quad / \quad \text{\textit{write(}'\text{,}'\text{), */ no. of indentation spaces */}}
\]

\[
p(\text{BdDfl, NewI}), \quad / \quad \text{\textit{write(}'\text{end.}'\text{), */ specification body */}}
\]

\[
\text{\textit{write(]'\text{,}'\text{), */ end of specification */}}
\]

occurs. When the nodes that correspond to terminal symbols are reached these terminals are printed out. Care has also been taken to pretty-print the output specification using indentation (variables I and NewI) above. Each time a new or nested block of statements is introduced the tab variables I and NewI (as they are referred to throughout the printing module) are changed to reflect the nesting level. The size of the Prolog program implementing the printing routines is 662 lines.

Table 3.3 shows the results of performance measurements of the printing routines. The input is the syntax tree of the normalized specification (measured in bytes) and the output is the normalized specification. The average system load during printing was 0.30 Erlangs and the experiment was conducted ten times for each protocol.
<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input syntax tree (bytes)</th>
<th>number of transitions before</th>
<th>number of transitions after</th>
<th>size of output syntax tree (bytes)</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>12,267</td>
<td>5</td>
<td>9</td>
<td>14,840</td>
<td>1.522</td>
</tr>
<tr>
<td>simple transport map module</td>
<td>16,438</td>
<td>20</td>
<td>37</td>
<td>22,802</td>
<td>12.878</td>
</tr>
<tr>
<td>simple transport map module</td>
<td>17,358</td>
<td>5</td>
<td>53</td>
<td>159,816</td>
<td>21.033</td>
</tr>
<tr>
<td>FTAM</td>
<td>81,267</td>
<td>36</td>
<td>103</td>
<td>99,958</td>
<td>88.105</td>
</tr>
<tr>
<td>LAP-D</td>
<td>167,418</td>
<td>128</td>
<td>475</td>
<td>692,686</td>
<td>52.399</td>
</tr>
</tbody>
</table>

Table 3.2. Performance of the normalization module

<table>
<thead>
<tr>
<th>normalized specification</th>
<th>size of input syntax tree (bytes)</th>
<th>size of output specification (lines)</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>15,084</td>
<td>271</td>
<td>1.175</td>
</tr>
<tr>
<td>simple transport</td>
<td>226,886</td>
<td>2,071</td>
<td>14.342</td>
</tr>
<tr>
<td>FTAM</td>
<td>145,248</td>
<td>2,332</td>
<td>8.650</td>
</tr>
<tr>
<td>LAP-D</td>
<td>691,249</td>
<td>8,741</td>
<td>49.703</td>
</tr>
</tbody>
</table>

Table 3.3. Performance of the printing module
CHAPTER 4

DATA AND CONTROL FLOW ANALYSIS

After having normalized the input specification two types of flow are specified, [18]:

- **Flow of data** which shows how operations in the actions part of a transition are applied on the input interaction parameters or context variables in order to determine the value of the output interaction parameters, and

- **Flow of control** which shows the major state changes (i.e. the finite state machine implemented by an Estelle specification).

This chapter explains how this information is collected and stored in a form suitable for processing by graphics tools.

4.1. DATA FLOW ANALYSIS

Data flow analysis consists of two phases. The input to the first phase is a normalized specification. The output is an equivalent specification such that all the PDUs exchanged in interactions or processed in transition blocks are explicitly identified. The second phase processes the output of the first phase, determines the nodes of the data flow graph and prints them in a form suitable for a graphics tool.

Examples are extracted from normalized specifications of single module alternating bit, [11], or two module transport protocol, [3], except when the normal form structure and the original one are the same.
4.1.1. Data Flow Analysis - First Phase

In this phase the declaration and transition declaration parts are processed separately. Declaration part processing is intended to expand the declarations referring to PDUs in order to meet the symbolic replacement of identifiers, which takes place in transition declaration part processing.

Knowledge of the kind of PDU exchanged in an interaction is important during the partitioning of data flow graph into protocol functions.

4.1.1.1. Declaration Part Processing

Four types of Estelle structures are processed in this module:

a. Variant records,

b. Channel definitions,

c. Variable declarations,

d. Primitive procedure and function declarations.

A case where variant records are frequently used is the declaration of PDUs. Each PDU is identified using a tag-field. Consider for example the record definition

\[
\text{TDPUCodTp} = (\text{CR}, \text{CC}, \text{DR}, \text{DC}, \text{DT}, \text{AK}, \text{undef\_code});
\]

\[
\text{TDPUnCollInf} = \text{record}
\]

\[
\begin{array}{ll}
\{ \text{control information} \} \\
\text{full} & : \text{boolean}; \\
\text{order} & : \text{orderTp}; \\
\text{peerAddr} & : \text{TAddrTp}; \\
\end{array}
\]

\[
\{ \text{fields of TDPU} \} \\
\text{crVal} & : \text{creditTp}; \\
\text{destRef} & : \text{refTp}; \\
\text{SrcRef} & : \text{refTp}; \\
\text{user\_data} & : \{ \text{optional} \} \text{dataTp}; \\
\end{array}
\]

\[
\begin{array}{ll}
\{ \text{used for CR, CC, AK} \} \\
\{ \text{used for CC, DR, DC, DT (class 2 only), EDT, AK, EAK, ERR} \} \\
\{ \text{used for CR, CC, DR, DC} \} \\
\{ \text{see TSAP; used for CR, CC, DR (not in this version of the protocol), DT, EDT} \}
\end{array}
\]

\[
\text{case kind : TDPUCodTp of}
\]
Tag-field kind is used to identify seven possible types of PDUs, namely CR, CC, DR, DC, DT, AK and undef_code.

The first phase of data flow analysis determines which fields belong to a certain PDU type and creates a separate record definition for each type. Clearly, all the fields appearing in the fixed part of the record declaration should be present in the record definition of each PDU. Thus, the fields which are used by each PDU type are explicitly defined. Additionally, one more data type containing all the fields of fixed and variant part and the tag-field as a common record field is produced (i.e. case is removed). The reason is that some data structures may hold more than one kind of PDU at the same time (e.g. a buffer). In this case the data structure must be defined as the union of all the PDU data types. A case where this is used is presented in section 4.1.1.2. The record definition yielded for CR PDUs is

```plaintext
tpdualndctrlinf_cr =
  record
    full: boolean;
    order: ordertp;
    peeraddr: taddrtp;
    ercol: credittp;
    destref: reftp;
    srceref: reftp;
    user_data: datatp;
    opts_ind: opttp;
    tsapid_calling, tsapid_called: t_sufTp
  end;
```
Tags in PDU variant records can be repeated instead of repeating the same fields for different PDU types (which is not semantically correct). In this case all the fields belonging to a PDU type are collected and then a record definition for this PDU is made. In record definition

\[
\text{rec} = \text{record} \\
\begin{array}{l}
  f1: \text{integer}; \\
  f2: \text{boolean}; \\
  \text{case Tag : TagType of} \\
  t1: (f3: \text{integer}); \\
  t2, t1: (f4: \text{char}); \\
  t3, t1: (f5: \text{any\_type}); \\
\end{array}
\]

end;

tag \text{t1} is repeated (tag-field is a scalar type \text{TagType} = (t1, t2, t3)). Then data flow analysis produces the records

\[
\begin{align*}
\text{rec}_3 = & \text{record} \\
& f1: \text{integer}; \\
& f2: \text{boolean}; \\
& f5: \text{any\_type} \\
\text{end}; \\
\text{rec}_1 = & \text{record} \\
& f1: \text{integer}; \\
& f2: \text{boolean}; \\
& f3: \text{integer}; \\
& f4: \text{char}; \\
& f5: \text{any\_type} \\
\text{end}; \\
\text{rec}_2 = & \text{record} \\
& f1: \text{integer}; \\
& f2: \text{boolean}; \\
& f4: \text{char} \\
\text{end};
\end{align*}
\]

There is no restriction concerning the case constants of a variant record but when this record is used for PDU definition the case constants (i.e. PDU identifiers) should not be signed numbers, since this will introduce errors when the record name and case constant name are concatenated.

The enumeration of PDU variant records affects other Estelle structures which use parameters of PDU record type. These types are \text{channels}, \text{variables}.
and primitive procedures and functions.

When an interaction in a channel definition carries PDUs it is expanded so that the type of the PDU exchanged in this interaction is explicitly specified. For example, the channel definition

```
CHANNEL PDUandCtrlPrims(protocol, mapping);
  BY protocol, mapping :
    transfer (PDU : TPDUandCtrlInf);
    term;
  BY mapping:
    ready; { ready for one more block }
{ end of PDUandCtrlPrims }
```

is expanded to the channel definition

```
channel pduandctrlprims(protocol, mapping);
  by protocol:
    transfer_cc(pdu : tpduandctrlinf_cc);
    transfer_cr(pdu : tpduandctrlinf_cr);
    transfer_dr(pdu : tpduandctrlinf_dr);
    transfer_dc(pdu : tpduandctrlinf_dc);
    transfer_dt(pdu : tpduandctrlinf_dt);
    transfer_ak(pdu : tpduandctrlinf_ak);
    transfer.Undef_code(pdu : tpduandctrlinf.Undef_code);
    term;
  by mapping:
    transfer_cc(pdu : tpduandctrlinf_cc);
    transfer_cr(pdu : tpduandctrlinf_cr);
    transfer_dr(pdu : tpduandctrlinf_dr);
    transfer_dc(pdu : tpduandctrlinf_dc);
    transfer_dt(pdu : tpduandctrlinf_dt);
    transfer_ak(pdu : tpduandctrlinf_ak);
    transfer.Undef_code(pdu : tpduandctrlinf.Undef_code);
    term;
    ready;
```

Notice that each role and its interaction definitions has been listed separately (the shorthand notation followed by the original specification has been expanded).

Variable declarations referring to PDUs (e.g. variables of type TPDUandCtrlInf) are also expanded. Data flow analysis of the variable declaration

```
recPDU: TPDUandCtrlInf;
```
produces a number of declarations:

```
recepdu_undef_code: tpdualandctrlinf_undef_code;
recepdu_ak: tpdualandctrlinf_ak;
recepdu_dt: tpdualandctrlinf_dt;
recepdu_dc: tpdualandctrlinf_dc;
recepdu_dr: tpdualandctrlinf_dr;
recepdu_cr: tpdualandctrlinf_cr;
recepdu_cc: tpdualandctrlinf_cc;
```

Prefixes `_undef_code`, `_ak`, `_dt`, `_dc`, `_dr`, `_cr` and `_cc` are used to denote the PDU type carried by each of the new variables.

Next procedure and function declarations with directives `primitive`, `external`, or `forward` are handled. This processing is done after the data flow analysis of transitions is completed. When a procedure/function call is found inside a transition block, it is checked whether any of the formal parameters or the function itself are of PDU type (e.g. of type `TPDUandCtrlInf`). In such a case all the necessary information is stored in a Prolog clause and new procedure/function declarations are derived in the following way:

a. formal parameters referring to input PDUs are redefined to be of the record type defining the kind of PDU in hand,

b. other formal parameters are redefined to be of the same type as the type of PDU being processed inside transition block,

c. data type of PDU under processing becomes the type of function and

d. the names of input PDU and PDU under processing are appended to the name of procedure/function in order to form the new declaration.

Assume, for example, that procedure `process_PDU` is defined as

```
procedure process_PDU(in_PDU: TPDUandCtrlInf;
                       var out_PDU: TPDUandCtrlInf);
primitive;
```

and when it is called `in_PDU` is assigned an input parameter referring to a `CR` PDU and `out_PDU` should return a `CC` PDU. Then data flow analysis produces
procedure process_pdu_cr_cc(in_pdu: t pdu andctrlinf_cr;
    var out_pdu: t pdu andctrlinf_cc);

Clearly, this process results in a minimum number of declarations. Each time a
call to a routine with a new permutation of PDU kinds occurs, a new declaration
is created. In the previous example, if all possible PDU kinds had been con-
sidered (i.e. seven PDUs) 49 procedure declarations would have been created even
if only one was necessary.

4.1.1.2. Transition Declaration Part Processing

The transition declaration part must also be changed: PDUs exchanged as
interaction parameters must be identified and the interaction names must change
accordingly. Data flow analysis is applied on normal form Estelle transitions in
order to define the kind of PDUs in

- Input Interactions and
- Transition Block and Output Interactions.

The next step is the replacement of variable and interaction names with names
denoting explicitly the PDUs carried. The declarations defining variables and
interactions referring to PDUs are created during the declaration part processing
of normalization (3.2.1).

In order to determine the kind of incoming PDUs (when clause) the provided
clause of a transition is scanned. This may indicate the PDU expected as
an input. If this scanning is successful the interaction name in the when clause
is changed so that it reflects the kind of PDU exchanged. For example, the provided
clause in the transition

```
WHEN Map.transfer
  PROVIDED PDU.kind = DR
  FROM waitCC
  TO closed
  begin
```
OUTPUT TS.TDISind(PDU.disc_reason);
OUTPUT Map.term;
end;

implies that the PDU expected in the input is of kind DR, otherwise the transition cannot be fired. The result of data flow analysis is

trans
{ 04 }
when map.transfer_dr
provided true
from waitcc
to closed
begin
  output is.tdisind(pdu.disc_reason);
  output map.term
end;

The boolean expression in the provided was replaced by true since the interaction transfer_dr carries a DR PDU.

It should be noted that it is possible that the kind of PDU expected does not appear in the provided clause of the original transition (e.g. when more than one PDU kind are handled). In this case there are two possibilities:

a. There is some reference to PDU kinds in conditional statements inside the body of the transition. The conditions must imply the PDU kind excluding all other kinds, otherwise ambiguities may be introduced. Normalization module moves these conditions to the provided clause generating a new transition for each path in the body of the original transition (see 3.2). Therefore, data flow analysis can determine the PDU kind for each normal form transition.

b. If the kind of PDU exchanged in an input interaction cannot be determined from the provided clause of the normal form transition,

- a new transition for each PDU kind is generated and
- assignment statements assigning the input PDU to a variable of data type designed to hold more than one kinds of PDU at the same time
(e.g. a buffer) are replaced by a sequence of assignment statements assigning each field of the input PDU to the corresponding field of the variable. The original declaration of this variable must be PDU record type. Since variant PDU records are enumerated the new type must be a union of all the enumerations. Also the tag-field of the case structure of the original PDU record type must be included in the new definition in order to define what kind of PDU is in the buffer.

Consider the transition,

```
TRANS
ANY T_suf : T_sufTp; EPId : TCEPIdTp DO
WHEN AP[T_suf, EPId].transfer { PDU }
  { this input may occur with any value of T_suf, EPId }
  begin
     TC[T_suf, EPId].PDU_buf[PDU.kind] := PDU;
     with TC[T_suf, EPId].PDU_buf[PDU.kind] do
       begin
          full := true;
       end
  end;
```

where TC is an array of record keeping information about local and remote references, network connections and incoming PDUs. The latter are put in an array of type `TPDUandCtrlInf` named `PDU_buf`. The definition of `PDU_buf` should not be expanded. Data flow analysis produces seven transitions, two of which are

```
trans
  { 1 }
any t_suf: t_sufTp; epid: tcepidTp do
provided true
when ap[t_suf, epid].transfer_undef_code
from idle
to idle
begin
  tc[t_suf, epid].pdu_buf[undef_code].user_data := pdu.user_data;
  tc[t_suf, epid].pdu_buf[undef_code].srcref := pdu.srcref;
  tc[t_suf, epid].pdu_buf[undef_code].destref := pdu.destref;
  tc[t_suf, epid].pdu_buf[undef_code].crvl := pdu.crvl;
  tc[t_suf, epid].pdu_buf[undef_code].peeraddr := pdu.peeraddr;
  tc[t_suf, epid].pdu_buf[undef_code].order := pdu.order;
  tc[t_suf, epid].pdu_buf[undef_code].full := pdu.full;
```
\[
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{undef}_\text{code}], \text{kind} := \text{undef}_\text{code};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{undef}_\text{code}], \text{full} := \text{true}} \\
\text{end;}
\]

\[\text{trans}\]
\{ 7 \}
\text{any } t_{su foulp; epid: tcepdtp do}
\text{provided true}
\text{when ap[t_{su}f, epid].transfer_cc}
\text{from idle}
\text{to idle}
\text{begin}
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{tsapid}_\text{called} := \text{pdu}.\text{tsapid}_\text{called};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{tsapid}_\text{calling} := \text{pdu}.\text{tsapid}_\text{calling};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{opts}_\text{ind} := \text{pdu}.\text{opts}_\text{ind};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{user}_\text{data} := \text{pdu}.\text{user}_\text{data};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{src}_\text{ref} := \text{pdu}.\text{src}_\text{ref};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{dest}_\text{ref} := \text{pdu}.\text{dest}_\text{ref};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{crl} := \text{pdu}.\text{crl};} \\
\text{tc}[_\text{su}f, \text{epid}, \text{pdu}_\text{buf}[\text{cc}], \text{peeradd}_\text{dr} := \text{pdu}_\text{pdu}_\text{buf}[\text{cc}].\text{order} := \text{pdu}_\text{pdu}_\text{buf}[\text{cc}].\text{full} := \text{true} \\
\text{end;}
\]

Similar transitions are created for Input Interactions for AK, DT, DC, Dh and CR.

Next transition block and output Interactions are processed.

If a variable refers to a PDU,

- the PDU kind is determined from a statement assigning the PDU kind to the variable’s tag-field or, if an Input PDU exists, from a statement assigning the input PDU to this variable,

- the statement assigning the PDU kind to the variable’s tag-field is removed and

- variable’s name is extended with PDU’s kind.
Variable pdu_cc_pdu in the transition

\[
\begin{align*}
\text{trans} & \{ 06 \} \\
& \text{when ts.tcorresp} \\
& \text{provided acptdopts} \leq \text{opts} \\
& \text{from waittconresp} \\
& \text{to open} \\
& \text{begin} \\
& \text{opts} := \text{acptdopts}; \\
& \text{trseq} := 0; \\
& \text{tseq} := 0; \\
& \text{pdu}_\text{cc}_\text{pdu}.\text{kind} := \text{cc}; \\
& \text{pdu}_\text{cc}_\text{pdu}.\text{opts}_\text{ind} := \text{opts}; \\
& \text{pdu}_\text{cc}_\text{pdu}.\text{crvl} := \text{rer}; \\
& \text{pdu}_\text{cc}_\text{pdu}.\text{order} := \text{first}; \\
& \text{cc}_\text{pdu}_\text{10} := \text{pdu}_\text{cc}_\text{pdu}; \\
& \text{output map.transfer}(\text{cc}_\text{pdu}_\text{10}) \\
& \text{end}; \\
\end{align*}
\]

refers to CC PDU. Any reference to this variable is replaced by pdu\_cc\_pdu\_cc
which is declared of type tptuandcttrlinf\_cc and defines CC PDUs only:

\[
\begin{align*}
\text{trans} & \{ 06 \} \\
& \text{when ts.tconresp} \\
& \text{provided acptdopts} \leq \text{opts} \\
& \text{from waittconresp} \\
& \text{to open} \\
& \text{begin} \\
& \text{opts} := \text{acptdopts}; \\
& \text{trseq} := 0; \\
& \text{tseq} := 0; \\
& \text{pdu}_\text{cc}_\text{pdu}_\text{cc}.\text{opts}_\text{ind} := \text{opts}; \\
& \text{pdu}_\text{cc}_\text{pdu}_\text{cc}.\text{crvl} := \text{rer}; \\
& \text{pdu}_\text{cc}_\text{pdu}_\text{cc}.\text{order} := \text{first}; \\
& \text{cc}_\text{pdu}_\text{10}\_\text{cc} := \text{pdu}_\text{cc}_\text{pdu}_\text{cc}; \\
& \text{output map.transfer}_\text{cc}(\text{cc}_\text{pdu}_\text{10}\_\text{cc}) \\
& \text{end}; \\
\end{align*}
\]

If the PDU kind cannot be determined, the user is informed with message

\[
\text{line L: failed to determine PDU kind referred to by variable V,}
\]

where \( L \) is the line number and \( V \) the variable name. In such a case the variable remains unchanged in the output of this phase.

Next, the names of output interactions carrying PDUs are considered. The
output interaction name changes according to the kind of its PDU parameters. The actual parameters are variables whose kind is found in the previously described way. Output interaction \textit{transfer(cc_pdu_10)} in the previous example carries a \textit{CC} PDU. After data flow analysis the interaction becomes \textit{transfer\_cc(cc_pdu_10\_cc)} which is declared to have as parameters \textit{CC} PDUs only (section 3.2.1). If the kind of output PDU cannot be determined, the user is informed with message

\textit{line L: failed to determine PDU kind referred to in interface event Event,}

where \textit{L} is the line number and \\textit{Event} the interaction name.

The current implementation can handle more than one PDU kinds per transition block. That is, different variables can be of different PDU type. Also, it is not necessary to specify the PDU kind corresponding to a variable in the beginning of the transition block. This could be done anywhere inside the transition block.

4.1.1.3. Implementation of First Phase of Data Flow Analysis

The data flow analysis module searches the parse tree of the normalized specification in a depth-first way. It analyzes each part of the specification producing a new subtree for this part. Finally, a new syntax tree is created and printed as an Estelle specification using the printing module (see 3.3.1).

First, the declaration part is processed. The PDU variant records are expanded by the clauses

\[\text{normRec}(\text{TpName}, \text{fldLst}(\text{VrntPrt}), \text{TpDfs}) : - \]
\[\text{assert(pduItcrdFzdPrt(\text{empty}))} \quad \text{/* record fixed part is empty */} \]
\[\text{getCsCnstFlds(\text{VrntPrt}, \text{.})} \quad \text{/* get case constants from variant part */} \]
\[\text{recDf(\text{TpName}, \text{TpDfs}), !} \quad \text{/* make new record defs for each PDU kind */} \]

\[\text{normRec}(\text{TpName}, \text{fldLst}(\text{FzdPrt}, \text{VrntPrt}), \text{TpDfs}) : - \]
assert(pduRcrdFzdPrt(FzdPrt)),  /* record fixed part is not empty */
getCsCnstFlds(VrtPrtr),  /* get case constants from variant part */
recDf(TpName, TpDfs), !.  /* make new record defs for each PDU kind */

When channel declarations with interactions carrying PDUs are encountered
the clauses mkChBlck/2 expand the channel block enumerating interaction
definitions. The channel definition and its roles are obtained from the dictionary:

mkChBlck(N, ChBlck) :-
    /* get definition of channel N from dictionary (N1, N2 are roles) */
    find(N, [channel, [N1, IntDfs1], [N2, IntDfs2]]),
    mkIntGr(N1, IntDfs1, IntGr1),  /* make interaction group of role N1 */
    mkIntGr(N2, IntDfs2, IntGr2),  /* make interaction group of role N2 */
    ChBlck =  /* make channel block */
    chBlck(chBlck(IntGr1, IntGr2), !.  /* from interaction groups */

Variable declarations referring to PDUs are expanded by the clause

dt1(vrDcl(IdLst, tpDntr(smplTp(id(PduT, _))))), VrDcls) :-
pduTp(PduT),  /* PDU type */
    pduLst(PduL),  /* get list of PDU kinds */
    getIds(IdLst, IdL),  /* get a Prolog list of variables */
    mkVrDcls(PduL, IdL, PduT, VrDcls), !.  /* make new variable declarations */

Primitive function or procedure declarations are handled by the clauses
mkPrCdf/1 and mkFncDf/1. Since clause assertion was used to store information
about the input parameters, the input PDU type, the PDU kind processed by the
transition block and the function name, the new new declaration is made by
retracting the clauses keeping this information (prcDf/4 and fncDf/4) and build-
ing a procedure/function declaration subtree from prcDf/4's or fncDf/4's con-
tents:

mkPrCdf(dclPrt(DclPrt,
    dcls(prcDcl(prcHd(procedure, id(N, 0), FrmlPrlmLst), BdTp)))) :-
    retract(prcDf(N0, K, InpPrms, InPduTp)),
    find(N0, [procedure, [BdTp], PrmlLst0]),
    reverse(PrmlLst0, [], PrmlLst),
    conc(N0, K, N1),
    conc(N1, InPduTp, N),
    mkFrmlPrlmLst(PrmlLst, FrmlPrlmLst, K, InpPrms, InPduTp),
    mkPrCdf(DclPrt), !.
mkPrcDf(dclPrt) :- !.

mkFncDf(dclPrt(DclPrt,
    dcls([fncDc([fncHd(function, id(N, 0), FncFrmPrmLst, id(T, 0)), BdTp])) :-
      retract([fncDc([N0, K, InpPrms, InPduTp]),
        find([N0, [[function, [Type, BdTp]], PrmLst0]],
          reverse(PrmLst0, [], PrmLst),
            (pduTp(Type),
              conc(Type, K, T)),
              T = Type),
              conc(N0, K, N1),
              conc(N1, InPduTp, N),
              mkFncFrmPrmLst(PrmLst, FncFrmPrmLst, K, InpPrms, InPduTp),
              mkFncDf(DclPrt), !.
              mkFncDf(dclPrt) :- !.

The clause processing the input interactions, if the PDU kind can be determined from the provided, is

proWhenAnaProv(ClS0, Cls) :-
  getInInt(ClS0),
  proProv(ClS0, Cls1, Vr),
  proWhen(ClS1, Cls2),
  inpPrms(Prms),
  ((member([Vr]), Prms),
    Cls = Cls2),
  (@vK(K),
    conc(Vr, K, NewVr),
    repNm(Vr, NewVr, Cls2, Cls))),
  retractall(pdu_kind(_)), !.
/* get input interactions from clauses */
/* get PDU kind. Vr = PDU variable */
/* change input interaction if provided */
/* implies input PDU kind */
/* Get parameters of input interaction */
/* If Vr is input parameter don't */
/* replace its name; */
/* otherwise */
/* concatenate PDU kind to var name */
/* and replace name in the clauses */
/* Remove information about PDU kind */

If the kind of input PDU cannot be determined clause expandWhen/2 generates one transition for each PDU kind:

expandWhen1(trGr(ClS0, trBlck(Cdp, Tdp, Vdp, Pafdp, Tn, cmpStmt(StmtSeq))),
  TrGr) -
  proProv(ClS0, _),
  not retractall(pdu_kind(_)),
  retractall(inInt(_)),
  getInInt(ClS0),
  pduTp(Tp),
  retract(inInt(_, Prms)),
  member([Pdu, [parameter, [Tp]], Prms],
    pduLst(TagLst),
    pdu_id_kind(K),
    / * If PDU kind is not implied by */
    / * provided (retract/1 fails) then */
    / * remove old information */
    / * get new information */
    / * get name of PDU record */
    / * check if any of the input */
    / * parameters is PDU */
    / * get the list of PDU kinds */
    / * get the name of the tag-field */
expandTrans(Cls0, Pdu, K, TagList, StmtSeq, PathList), /* make a path for each PDU kind */
mkTr(PathList, [Cop, Tdp, Vdp, Pafdp, Tn], TrGrs), /* Make one trans for each path and */ /* link them in a new set */ /* of transition groups */

When the transition block is processed, clause getPduKind/2 determines the kind of PDU being processed by examining each assignment statement:

getPduKind(stmt(VrAcc0, Xpr), stmt) :-
  lastFlfSpc(VrAcc0, Name1), /* If the tag-field of a PDU variable */
  pdu_id_kind(Name1), /* is being assigned */
  varRefToPdu(VrAcc0, _, _), /* If VrAcc0 refers to a FDU */
  value(Xpr, Vl), /* Examine the value of expression Xpr: */
  pduList(L), /* the value of Xpr is Vl */
  member(Vl, L), /* L is list of PDU kinds */
  retv:actall(pdu_kind(_)), /* if the value Vl of Xpr denotes PDU kind */
  assert(pdu_kind(Vl)), !. /* remove old info about PDU kind */
  /* store new info about PDU kind */

Concerning the modifiability of the data flow analysis - phase I - module the same remarks as the ones stated in section 3.3 hold: Any modifications to the Estelle syntax rules should be reflected to the syntax tree structure and subsequently to the clauses manipulating this tree.

4.1.1.4. Performance of the First Phase of Data Flow Analysis.

The performance experiments were executed under an average CPU load of 1.36 Erlangs. Each protocol was processed ten times and the resulting times were averaged. Table 4.1 shows the results of the experiments. The input and output syntax trees represent the data processed and generated, respectively. The processing time is proportional to the number of transitions in the input and is not related strongly to the increase of this number. The size of the programs implementing the first phase of data flow analysis is 1045 lines.

4.1.2. Data Flow Analysis - Second Phase

The input to the second phase of data flow analysis is a normalized Estelle
<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input syntax tree (bytes)</th>
<th>number of transitions before</th>
<th>number of transitions after</th>
<th>size of output syntax tree (bytes)</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>14,840</td>
<td>9</td>
<td>9</td>
<td>15,084</td>
<td>3.461</td>
</tr>
<tr>
<td>simple transport ap module</td>
<td>22,802</td>
<td>37</td>
<td>37</td>
<td>29,327</td>
<td>17.417</td>
</tr>
<tr>
<td>simple transport map module</td>
<td>159,816</td>
<td>53</td>
<td>59</td>
<td>179,394</td>
<td>150.683</td>
</tr>
<tr>
<td>FTAM</td>
<td>99,958</td>
<td>103</td>
<td>103</td>
<td>145,248</td>
<td>110.911</td>
</tr>
<tr>
<td>LAP-D</td>
<td>692,686</td>
<td>475</td>
<td>530</td>
<td>691,249</td>
<td>587.321</td>
</tr>
</tbody>
</table>

Table 4.1. Performance of data flow analysis - phase I

specification produced by the normalization (section 3.2) and the first phase of data flow analysis (section 4.1.1). This specification may need some modifications by the user if the first phase of data flow analysis failed to identify the PDU kind in an interaction or carried by a variable. The second phase calls the compiler module described in section 3.1 in order to check the normalized specification against syntax and semantic errors and produce a syntax tree and a dictionary as Prolog clauses. The output is a model of the information flow in the Estelle specification, excluding the major state changes, [21], and it is printed as a transition table. There are four kinds of data flow graph nodes:

- **I-nodes** representing input primitives,
- **D-nodes** representing data (variables and constants),
- **F-nodes** representing operations on data (i.e. functions, procedures and
operators) and

- **O-nodes** representing output primitives.

Information about node types and arcs of the data flow graph is printed in a file in the following form:

```
%nodes
node-name1 node-type1 data-type-of-node1
node-name2 node-type2 data-type-of-node2

%arcs
source-node1 destination-node1 transition-number1
source-node2 destination-node2 transition-number2
```

The arcs emerge from nodes representing data sources (e.g. I-nodes, D-nodes corresponding to constants) and are directed towards data sinks (e.g. O-nodes). D-nodes modeling actual parameters of a **procedure** or **function** call (modelled by an F-node) are connected with the F-node by

- a unidirectional arc directed towards the F-node if the parameters are called by value or
- a bidirectional arc if the parameters are called by reference (**var** parameters).

The data type of each node is used by another tool (**dfgtool**, [23]) for merging blocks of the data flow graph. The transition

```trans
def

begin
  copy(q.msgdata, n.data.data);
  b.ack.seq := n.data.seq;
  b.ack.conn := conn_end_p1_id;
  empty(b.ack.data);
  output n.data_squest_ack(b.ack)
end;
```

corresponds to a data flow graph described by the transition table (**arcs**
section)

21@copy q.msgdata 7
q.msgdata 21@copy 7
data_response_dt.ndata.data 21@copy 7
data_response_dt.ndata.seq b_ack.seq 7
22@empty b_ack.data 7
b_ack.data 22@empty 7
b_ack.conn data_request_ack.ndata.conn 7
b_ack.seq data_request_ack.ndata.seq 7
b_ack.data data_request_ack.ndata.data 7

where number 7 denotes the transition number and the nodes appearing are declared (%nodes section) as

b_ack.conn 2 subrange
b_ack.seq 2 subrange
b_ack.data 2 unspecified
q.msgdata 2 unspecified
q.msgseq 2 subrange
data_response_dt.ndata.seq 1 subrange
data_response_dt.ndata.data 1 unspecified
data_request_ack.ndata.conn 4 subrange
data_request_ack.ndata.seq 4 subrange
data_request_ack.ndata.data 4 unspecified
21@copy 8 procedure
22@empty 8 procedure

Numbers 1, 2, 4 and 8 declare I-nodes, D-nodes, O-nodes and F-nodes respectively. Prefixes n@, where n is a number, are used to name each F-node uniquely. This is done in order to distinguish among calls of the same function, procedure or operation in different transitions. When these operators are displayed the prefix is removed. Thus one F-node per operation is printed.

Special processing is done for two Estelle structures:

- **Arrays** and
- **any** clauses.

If the **array** elements are used as regular context variables then access to an **array** element is shown in the data flow graphs as a procedure or function call. More specifically an assignment statement of the type
\[a[i] := b;\]
corresponds to the procedure call

\texttt{assign\_array(a, i \ b);}  

and an assignment statement of the type

\[b := a[i];\]
corresponds to the function call

\[b := \text{index\_array(a, i);}\]

Procedure \texttt{assign\_array} and function \texttt{index\_array} can be viewed as being declared as

\[
\text{procedure assign\_array(var a: any\_array\_type;}
\quad i: \text{any\_index\_type;}
\quad b: \text{any\_element\_type);}  
\quad \text{primitive;}  
\]

\[
\text{junction index\_array(var a: any\_array\_type; i: any\_index\_type);}  
\quad \text{primitive;}  
\]

Consider the statement sequence

\[
\text{example\_array[1]} := \text{retrieve(send\_buffer);}  
\quad b\_dt.data := \text{example\_array[1];}
\]

where \texttt{example\_array} has been defined as an array. Data flow analysis produces the transition table

\[
\begin{array}{c}
5@assign\_array \text{example\_array 1} \\
\text{example\_array 5@assign\_array 1} \\
6@1 5@assign\_array 1 \\
\text{send\_buffer 7@retrieve 1} \\
7@retrieve 5@assign\_array 1 \\
9@1 8@index\_array 1 \\
8@index\_array \text{example\_array 1} \\
\text{example\_array 8@index\_array 1} \\
8@index\_array b\_dt.data 1 \\
\end{array}
\]

which is displayed as the graph of figure 4.1.
Figure 4.1. Data flow graph representation of a reference to an array element

It is possible that an array is connected to a channel. This, for example, happens when the array’s indices are of type transport connection identifier (tc_lid in reference [22]) or transport suffix and end-point identifier (t_suffix and ep_lid in reference [3]). In this case only one node is created containing the full array name including the indices. For example, the nodes produced for array tc in the transition

\[
\text{trans}
\{ 3 \}
\text{any t_suffix; epid: tcepidtp do}
\text{provided true}
\text{when ap[t_suffix, epid].transfer_dt}
\text{from idle}
\text{to idle}
\text{begin}
\text{tc[t_suffix, epid].pdu_buf[dt].end_of_tsdu := pdu.end_of_tsdu;}
\text{tc[t_suffix, epid].pdu_buf[dt].sendseq := pdu.sendseq;}
\text{tc[t_suffix, epid].pdu_buf[dt].user_data := pdu.user_data;}
\text{end;}
\]

are

\[
tc[\text{ANYt_suffix, ANYepid}.pdu_buf[dt].end_of_tsdu 2 unspecified}
tc[\text{ANYt_suffix, ANYepid}.pdu_buf[dt].sendseq 2 unspecified}
tc[\text{ANYt_suffix, ANYepid}.pdu_buf[dt].user_data.l 2 unspecified}
\]

Notice that indices of array of ips ap are the same as in tc (i.e. there is a kind of isomorphism between tc and ap).
The any clauses are removed during the second phase and the variables in the domain list are treated as global variables. The keyword ANY is displayed with the variables in the domain list of the any clause to remind the user that one transition should be generated for each value of the variables introduced by the any clause. The reason for not repeating the transition is that it would clutter the data flow graph and make the graph’s manipulation complex. For example the variable reason in the any clause of the transition

```plaintext
trans
{ II }
any reason: reasontp do
provided reason <> ts_user_init
from open
to wait_for_dc
begin
  output ts.tdisind(reason);
  pdu_dr_pdu_dr.disc_reason := reason;
  pdu_dr_pdu_dr.is_last_pdu := false;
  pdu_dr_pdu_dr.order := destructive;
  dr_pdu_13_dr := pdu_dr_pdu_dr;
  output map.transfer_dr(dr_pdu_13_dr)
end;
```
corresponds to node ANYreason in figure 4.2.

4.1.2.1. Implementation of Second Phase of Data Flow Analysis

The syntax tree of a normalized Estelle specification is scanned in a top-down fashion. When a transition group is found the clause

```plaintext
dtf2(trGr(Cls, TrBck)) :-
/* process ANY clauses */
proAny(Cls, AnyVrs, GlobVrs),
retractall(inInt(_:_)),
/* assert name and parameters of transition's input interaction */
getInInt(Cls),
/* data_flow analysis of transition block considering variables in
ANY clause as global variables */
dtf2(TrBck, AnyVrs, GlobVrs), !.
```
calls the routines that analyze the input clauses (getInInt/1) and the transition block (dtf2/3).
Figure 4.2. Data flow graph representation of ANY clauses

Clauses `getInInt/1` find the input interaction and store its name, parameters and type (I-node) in Prolog clause `ionode/3`. The main clause is

```prolog
getInInt(cl(whnCl(intRef(intFntRef, id(Ie, _)))), IntPntRef = ... [id(Ipr, _)], 
  getIntEvPrms(Ipr, Ie, Prms), 
  (retract(ionode(Ie, Prms, 1)); true), 
  assert(ionode(Ie, Prms, 1)), 
  assert(inInt(Ie, Prms)), !.
/* Get the parameters of */ 
/* interface event Ipr */ 
/* and assert them in Prolog */ 
/* clause ionode/3 (O-node = 1) */ 
/* Assert the same info in clause */ 
/* inInt/2 (used by first phase) */
```

The transition block consists of a statement sequence. Each statement type is processed separately. The following clause processes the output statements:

```prolog
def2(stmt(outStmt(intRef(intPntRef(id(Ipr, _)), id(Ie, _)), XprLst)), N) :- 
  XprLst =.. [xprLst1], 
  getIntEvPrms(Ipr, Ie, FrmPrmLst), 
  (retract(ionode(Ie, FrmPrmLst, 4)); true), 
  assert(ionode(Ie, FrmPrmLst, 4)), 
  getActPrmLst(XprLst, ActPrmLst), 
  assert ArcLst(Ie, FrmPrmLst), 
/* Get interface event parameters */ 
/* and assert them in Prolog */ 
/* clause ionode/3 (O-node = 4) */ 
/* Get list of actuals of OUTPUT */ 
/* and assert arcs from actu- */
```
A procedure call is processed by the clause

\[
\text{dtf2}(\text{stmt}(\text{id}(\text{ProcNm}, \_), \text{XprLst}), N) :\text{-}
\begin{align*}
\text{getActPrmLst}(\text{XprLst}, \text{ActPrmLst}), \\
\text{mkUnique}(&\text{ProcNm}, Nm1, \_\text{node}, \_\text{procedure}), \\
\text{find}(\text{ProcNm}, \{}[\text{procedure, }\_, \_], \text{ FrmI} \text{PrmLst}]\}, \\
\text{assertArcs}(Nm1, \text{ FrmI} \text{PrmLst}, \text{ActPrmLst}, N), !. \\
\end{align*}
\]

/* procedure call */
/* get list of actuals */
/* make a unique name from procedure's name */
/* Assert arcs from actuals */
/* to formals (N=trans.no) */

An assignment statement can be one of the types:

a. variable assigned the value of a function (with or without parameters),

b. array element assigned an expression,

c. variable assigned the value of an expression containing an operator and

d. variable assigned the value of an array element.

Clearly, a. is a special case of d. In both cases there is an edge connecting the D-node representing the variable (or the variable's fields in case of record) with the F-node representing the function or the operator. The reason for treating them separately is the difference between the syntax of function calls and simple operations. An expression can always contain a function call. The recursive nature of dtf2 clauses (implementing the second phase of data flow analysis) ensures that any combination will be analyzed regardless of nesting level. The clauses handling the aforementioned cases of assignment statements are:

\[
\text{dtf2}(\text{stmt}(\text{VrAcc}, \text{xpr}(\text{VrAcc}(\text{id}(\text{Nm0}, \_))))), N) :\text{-}
\begin{align*}
\text{if} \ "\text{Nm0}" \text{ is a function identifier} &\text{ */} \\
\text{find}(\text{Nm0}, \{}[\text{function}, \_T])\}, \\
(T = [Tp]; T = [Tp, \_]), \\
\text{mkUnique}(\text{Nm0}, Nm1, \_\text{node}, Tp), \\
\text{get the full name of node in left side of assignment statement} \\
\text{and the list of its fields } &\text{ */} \\
\text{leftSideNode}(\text{VrAcc}, \text{Nm2}, \text{FldLst}), \\
\text{assert an arc from function's name to each field of record variable } &\text{ */} \\
\text{assertArcLst4}(N, Nm1, Nm2, \text{FldLst}), !.
\end{align*}
\]
/* variable assigned by a function with parameters */
dfl2(stmt(VrAcc, xpr(id(Nm0, _), XprLst)), N) :-
    /* if "Nm0" is a function identifier */
    find(Nm0, [function, T], FrmlPrlmLst),
    (T = [Tp, _]),
    /* make a unique name from function's name */
    mkUnique(Nm0, Nm1, fnode, Tp),
    /* get the full name of node in left side of assignment statement and the list of its fields */
    leftSideNode(VrAcc, Nm2, FldLst),
    /* assert an arc from function's name to each field of record variable */
    assertArcLst4(N, Nm1, Nm2, FldLst),
    /* get function's actual parameter list */
    getActPrlmLst(XprLst, ActPrlmLst),
    /* assert arcs from actuals to formals (N = trans.no) */
    assertArrs(Nm1, FrmlPrlmLst, ActPrlmLst, N), !.

/* if an array element is assigned replace the assignment statement with a call to procedure 'assign_array' and analyze it in the usual way */
dfl2(stmt(vrAcc(vrAcc(id(Nm, L)), xprLst(Xpr1)), Xpr2), N) :-
    dfl2(stmt(id(assign_array, 0),
        xprLst(xprLst(xpr(vrAcc(id(Nm, L)))). Xpr1), Xpr2)), N), !.

/* if a variable is assigned an expression referring to: 
 - an operator (the expression may also include function designators) 
 - an array element (equivalent to function call 'index_array') 
 use 'getNodeNm' to get the name of the F-node in the right hand side of the expression. 'getNodeNm/Nm' also asserts arcs implied by expression Xpr but not directly connected with variable referred to by variable access VrAcc */
dfl2(stmt(VrAcc, Xpr), N) :
    getNodeNm(N, Xpr, Node1),
    /* get the complete name and the field list of the variable */
    leftSideNode(VrAcc, Node2, FldLst),
    /* assert arcs from Node1 to all fields (FldLst) of Node2 */
    assertArcLst2(N, Node1, Node2, FldLst), !.

As discussed earlier in this chapter and chapter three any changes to the syntax of Estelle affect the structure of the clauses manipulating the fragments of the syntax tree. The output of the second phase can be easily changed to meet the requirements of any data flow graph drawing tool: since the data flow information is stored in Prolog clauses the routines printing this information can be modified to produce any desirable format.
4.1.2.2. Performance of Second Phase of Data Flow Analysis

The performance experiments were conducted ten times under an average CPU load of 1.48 Erlangs. The resulting execution times were averaged and the outcome is given in table 4.2. The input is the specification's syntax tree and the output is data flow information expressed as a table of nodes and arcs. In the case of transport protocol and FTAM the number of nodes is greater than the number of arcs. The reason is that one node is created for each record field but not necessarily used if this field is not assigned a value explicitly. The unused nodes can removed by the graphics tool displaying the data flow. The size of the program implementing the second phase of data flow analysis is 700 lines.

<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input syntax tree (bytes)</th>
<th>output data flow information (bytes)</th>
<th>number of nodes created</th>
<th>number of arcs created</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>15,084</td>
<td>4,512</td>
<td>74</td>
<td>98</td>
<td>5.542</td>
</tr>
<tr>
<td>simple transport ap module</td>
<td>29,327</td>
<td>203,981</td>
<td>1,496</td>
<td>294</td>
<td>167.917</td>
</tr>
<tr>
<td>simple transport map module</td>
<td>179,394</td>
<td>64,882</td>
<td>2,755</td>
<td>1,674</td>
<td>505.333</td>
</tr>
<tr>
<td>FTAM</td>
<td>145,248</td>
<td>146,888</td>
<td>1,595</td>
<td>1,207</td>
<td>949.433</td>
</tr>
<tr>
<td>LAP-D</td>
<td>691,249</td>
<td>344,738</td>
<td>3,145</td>
<td>7,065</td>
<td>1499.67</td>
</tr>
</tbody>
</table>

Table 4.2. Performance of data flow analysis - Phase II
4.2. CONTROL FLOW ANALYSIS

Control flow analysis generates the transition table of the finite state machine corresponding to a normal form Estelle specification. The input to this module is the Prolog clause representation of syntax tree of a normalized specification. The output is printed in the form:

```
initial-state
from-state1 input1 list-of-outputs1 to-state1
from-state2 input2 list-of-outputs2 to-state2
...
```

This output is read by a graphics tool (cgtool, [23]) which displays the finite state machine on a Sun station as a graph (figure 4.3). For example, control flow analysis of the transition

```
trans
{ ? }
when n.data_response_dt

begin
...
output n.data_request_ack(b_ack)
end;
```

generates the transition table entry

```
7 ack_wait n.data_response_dt [n.data_request_ack] ack_wait
```

If a transition has no input or output interactions the word `nil` is printed in the place of the missing interaction.

4.2.1. Implementation of Control Flow Analysis

The syntax tree of a specification is searched in top-down fashion in order to identify the transition subtrees (i.e. the transition groups in Estelle terminology).

Then the clause

```
ctrl(trGr(Cls, trBlk(_, _, _, _, cmpStmt(StmtSeq)))) :-
  retract(transCount(N0)),  /* increase transition */
N is N0 + 1,            /* counter by 1 */
  assert(transCount(N)),  /* current transition is N */
```

Figure 4.3. Control flow graph of the alternating bit protocol

\[
\begin{align*}
\text{assert}(n/\text{TrNo}(N)), & \quad \text{/* assert normal form trans. no. */} \\
(\text{transInp}(\text{Cls}, N)), & \quad \text{/* get transition input, if input exists */} \\
\text{assert}(n/\text{TrInput}([N, \text{nil}]), & \quad \text{/* otherwise input is nil */} \\
\text{assertFrom}(\text{Cls}, N), & \quad \text{/* assert FROM state */} \\
\text{assertTo}(\text{Cls}, N), & \quad \text{/* assert TO state */} \\
\text{transOut}(\text{StmtSeq}, N), & \quad !. \quad \text{/* get output interactions */}
\end{align*}
\]

collects information about when, from and to clauses and output statements. This information is asserted as Prolog clauses which are called when the control flow analysis finishes in order to print out the transition table.

The modifiability is similar to modifiability of data flow analysis - phase II - module. Unless a change to the Estelle syntax imposes extensive modifications, it is very easy to output additional information by simply adding it to the clauses used for storing control flow information. Next, the programmer must change the routines outputting and formatting this information, i.e. the clause \text{writeC-trInfo/0} and the clauses it invokes.

4.2.2. Performance of Control Flow Analysis

The performance experiments were run ten times and the resulting execution times were averaged. The CPU load was 1.48 Erlangs. Table 4.3 shows the
outcome of the performance experiments. The input is the specification's syntax tree and the output control flow information is the transition table of the FSM described by the protocol specification. Control flow analysis is implemented in 290 lines of Prolog code.

<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input syntax tree (bytes)</th>
<th>output control flow information (bytes)</th>
<th>number of nodes created</th>
<th>number of arcs created</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternating bit</td>
<td>15,084</td>
<td>46</td>
<td>2</td>
<td>9</td>
<td>1.346</td>
</tr>
<tr>
<td>simple transport ap module</td>
<td>29,327</td>
<td>1,626</td>
<td>6</td>
<td>37</td>
<td>3.467</td>
</tr>
<tr>
<td>simple transport map module</td>
<td>179,394</td>
<td>2,418</td>
<td>1</td>
<td>59</td>
<td>6.533</td>
</tr>
<tr>
<td>FTAM</td>
<td>145,248</td>
<td>4,760</td>
<td>18</td>
<td>103</td>
<td>5.266</td>
</tr>
<tr>
<td>LAP-D</td>
<td>691,249</td>
<td>38,486</td>
<td>8</td>
<td>530</td>
<td>32.116</td>
</tr>
</tbody>
</table>

Table 4.3. Performance of control flow analysis
CHAPTER 5

MODULE MERGING

Module merging is the transformation of an Estelle specification containing two or
more modules to an equivalent specification of one module only. Test sequence
generation requires a single module representation of the protocol. The reason is
that interactions between modules of the same specification cannot be observed
and/or controlled by the tester of the protocol implementation, assuming black
box testing, [17].

It is assumed direct interaction between modules (rendez-vous) which is
referred to as intermodule communication and is carried over channels intercon-
necting children modules of a parent module. The assumption of rendez-vous
type of communication is essential. Estelle does not support this kind of com-
munication but this restriction can be worked out if the protocol specification
imposes only one message in the queues at any time. Theses messages can be
consumed by the modules immediately after they appear, thus simulating
rendez-vous communication. It is up to the protocol designer to decide whether
he/she will follow a specification style that satisfies this assumption. If rendez-
vous is not assumed, module merging results in a remarkable increase of the state
space since the contents of the channel queues should be considered as part of the
global state of the system.

This chapter explains how intermodule communication can be eliminated by
merging communicating modules of a protocol specification. The algorithm used
is called limited reachability analysis for extended finite state machines and was
first introduced in [15]. Reference [18] is an improved version of [15] and also
discusses implementation issues. Limited reachability analysis is intended to be used in validating protocol specifications. In this thesis, its use is restricted to module merging.

Module merging is applied on a normal form specification passed from data flow analysis - phase I and the resulting single module specification becomes input to data flow analysis - phase II and control flow analysis (figure 2.1). Module merging is an optional step of the specification analysis. If the user does not wish to have a merged specification or if the specification consists of only one module, he/she can skip the cmbn command (user's guide, appendix A) and go on with the dtf command for the control flow and the second phase of data flow analysis.

5.1. Combining Transitions in the Extended Finite State Machine Model

Each module of an Estelle specification can be represented by an EFMS. The product machine of two EFMSs is the EFMS produced by the merging of the two EFMSs. In module merging, a normal form transition of one module that produces an output to an internal channel (called combiner NFT) is combined with a normal form transition of the other module that consumes this output (called combinee NFT). The result is a transition of the product machine and is derived in the following way:

a. The state variable in the from clause is assigned an ordered pair consisting of the states in the from clauses of the combiner and the combinee NFT, respectively. The state variable in the to is created from the to states of the combined transitions in the same way.

b. The when clause of the combiner NFT (if present) becomes the when clause of the produced transition. This clause cannot refer to an internal
interaction (section 5.1.1).

c. The domain lists in the any clauses of the combined NFTs are concatenated to form the domain list of the produced NFT.

d. The boolean expressions in the provided clauses of the combined NFTs are anded. If the provided clause of the combinee NFT refers to an internal interaction parameter, the parameter is replaced by its actual value i.e. the expression corresponding to this parameter in the output statement of the combiner NFT.

e. The transition blocks of the combined NFTs are concatenated. The output in the combiner NFT which is consumed by the input of the combinee NFT is removed. The input parameters of the combinee NFT are replaced by the expressions assigned to those parameters by the output statement of the combiner NFT.

Consider, for example, the transitions

```plaintext
trans
{ 08 }
when ts.tdisreq
from open
to waitdc
begin
  pdu_dr_pdu_dr.discard_reason := ts_user_init;
  pdu_dr_pdu_dr.is_last_pdu := false;
  pdu_dr_pdu_dr.order := destructive;
  dr_pdu_12_dr := pdu_dr_pdu_dr;
  output map.transfer_dr(dr_pdu_12_dr)
end;

and

trans
{ 5 }
any t_suf; t_suf.tp, epid; tc.pidtp do
provided true
when ap[t_suf, epid].transfer_dr
from idle
to idle
begin
  tc[t_suf, epid].pdu_buf[dr].discard_reason := pdu.discard_reason;
```

The combined transition is

trans
any t_suf; t_suftp; epid: tcepidtp do
when ts.tdisreq
provided true
from open_idle
to waitdc_idle
begin
pdu_dr_pdu_dr.disc_reason := ts_user_init;
pdu_dr_pdu_dr.is_last_pdu := false;
pdu_dr_pdu_dr.order := destructive;
otherwise pdu_dr_12_dr := pdu_dr_pdu_dr;
tc[t_suf, epid, pdu_buf[dr].disc_reason := dr_pdu_12_dr.disc_reason;
tc[t_suf, epid, pdu_buf[dr].is_last_pdu := dr_pdu_12_dr.is_last_pdu;
tc[t_suf, epid, pdu_buf[dr].user_data := dr_pdu_12_dr.user_data;
tc[t_suf, epid, pdu_buf[dr].srcref := dr_pdu_12_dr.srcref;
tc[t_suf, epid, pdu_buf[dr].destref := dr_pdu_12_dr.destref;
tc[t_suf, epid, pdu_buf[dr].crvl := dr_pdu_12_dr.crvl;
tc[t_suf, epid, pdu_buf[dr].peeraddr := dr_pdu_12_dr.peeraddr;
tc[t_suf, epid, pdu_buf[dr].order := dr_pdu_12_dr.order;
tc[t_suf, epid, pdu_buf[dr].full := dr_pdu_12_dr.full;
tc[t_suf, epid, pdu_buf[dr].kind := dr;
tc[t_suf, epid, pdu_buf[dr].full := true
end;

Notice that the state values in the from and to clauses of the combined transitions are just concatenated because Estelle syntax does not permit ordered pairs as values of the state variables.

5.2. An Algorithm for Module Merging

The module merging algorithm is a modified version of the algorithm in [18]. The difference is that the new version applies to EFSMs. The modified algorithm is as follows.
Input:
Component EFSMs are EFSM1 and EFSM2;
EIPL is the external interaction point list and IIPL is the internal interaction point list used for communication between EFSM1 and EFSM2;

Output:
Combined EFSM.

Functions:
input(T[i]): returns null if transition T[i] is spontaneous, otherwise returns the input interaction of T[i] with the corresponding interaction point.
output(T[i]): returns the next output of transition T[i] with the corresponding interaction point or null if there is no more output left.

Output(point): returns the interaction point of an input or output interaction.

from(T[i]), to(T[i]): return the value of the state variable in the from or to clause of T[i].

Procedures:

combine_nft(T[i], T[j], T[i,j]): combine normal form transitions T[i] and T[j] in one transition T[i,j].
append_body(T[k], T[j]): append body of transition T[k] to T[j]'s body (modifies T[j]).

STEP 1:
for each transition T[i] in EFSM1 do
    if (input(T[i]) = null) or (T[i] in EIPL) then
        repeat
            out1 := output(T[i])
        if out1 <> null then
            \[\text{continue} \]
        else
            break
        end if
begin

for each transition T[j] in EFSM2 do

if ip(input(T[j])) = ip(output1) then

begin

combine_nft(T[i], T[j], T[i,j]);

tag the combined transition T[i,j] if T[j] has an
output to an internal interaction;

end

end

end

until out1 = null;

for each transition T[i] in EFSM2 do

{ interchange EFSM1 with EFSM2 and do the same processing as above }

...

for each tagged combined transition T[i,j] do

repeat

out1 := output(T[i])

if out1 in HPL then

begin

for each transition T[k] in EFSM1 or EFSM2 do

if (ip(input(T[k])) = ip(output1)) and (to(T[i,j]) = from(T[k])) then

begin

to(T[i,j]) := to(T[k]);

append_body(T[k], T[i,j]);

end

end

until out1 = null:
STEP 2:

for each transition $T[i]$ in EF$SM_1$ with no input or output to internal interaction points do

for state1 in States(EF$SM_2$) do

begin

add $T[i]$ to the list of combined transitions to be processed by STEP 3 by pairing its from and to states with state1;

end

for each transition $T[i]$ in EF$SM_1$ with no input or output to internal interaction points do

{ Interchange EF$SM_1$ with EF$SM_2$ and do the same processing as above }

...

STEP 3:

StateList := f;

for each combined transition $T[i,j]$ do

begin

StateList := StateList + from($T[i,j]$);

StateList := StateList + to($T[i,j]$)

end;

for each transition $T[i]$ output from STEP 2 do

if (from($T[i]$), to($T[i]$)) in StateList then

add $T[i]$ to the list of combined transitions

else

eliminate $T[i]$;

End of the Algorithm
If the specification contains more than two modules the algorithm can be applied initially on two modules, then the resulting module can be combined with a third module and so on.

5.3. Implementation

Clause assertion is used in order to store information needed throughout the processing of the modules to be merged. This information is:

- Interaction points connected via a `connect` statement are stored in clause `cnnStmt/2`. This information is needed for checking whether the input channel of a combinee NFT matches the output of the combiner NFT.
- The lists of external and internal interaction points are stored in clauses `eipl/1` and `iipl/1`, respectively.
- The names of the modules to be merged are stored in clauses `md(1, Md1)` and `md(2, Md2)`.
- The module body definition of each of the modules to be merged is stored in clause `mdbDdF/3`. This piece of information will be later used to get the transitions, the names of the module bodies to create the combined module body name and the declarations of each module body to put them in the combined module.

This preprocessing is performed by the `initialize/6` clause.

The `for` loops in the module merging algorithm are implemented with recursive depth-first searching of the subtree for transition declarations.

`Combine/6` is the main clause of the module merging program. Its structure follows strictly the structure of the algorithm in section 5.2:

```
combine(Tree, System, Md1, Md2, Eipl, Iipl):-
    initialize(Tree, System, Md1, Md2, Eipl, Iipl),
    step_1(List1, Md1, Md2),
    step_2(List2, Md1, Md2),
```
where List1, List2 are the transition lists output from step 1 and step 2, Comblist is the list of combined transitions and System represents the parent module of the combined modules. A new syntax tree is created from Comblist (variable NewTree), which is printed in a file named after the value of variable OutFile. The name of the combined module is produced by concatenating the names of the merged modules (variables Md1 and Md2).

A partial implementation of the module merging program existed at the time of development of this thesis. That implementation supported module merging of finite state machines considering only when clauses and output statements. A modification of the transition combination routines was applied in order to combine the statement sequences comprising the transition bodies. The first step of the algorithm checks each transition of one module against each transition of the other module to find out if a merging is possible. This is done by goal get_output/10 in get_trans1/7 clause:

get_trans1(trGr(Cls, trBlck(CnstDfPrt, TpDfPrt, VrDclPrt, PrcAndFncDclPrt, TrNm, cmpStmt(StmtSeq)),
               TrDclPrt2, Md1, Md2, List, 1, Done) :-
   Done is 0,
   ( find_when(Cls, Input), !, eipl(Input); /* spontaneous or */
     /* external source */
   TrGr = trGr(_, trBlck(CnstDfPrt, TpDfPrt, VrDclPrt, PrcAndFncDclPrt, TrNm, cmpStmt(_))),
   get_output(StmtSeq, TrDclPrt2, Md1, Md2, List, TrGr, Cls, StmtSeq, 1, _), !.

Clauses get_output/10 gets the output interaction in the statement sequence StmtSeq of a transition of the first module and tries to find a transition with a matching input in the transition declaration part TrDclPrt2 of the other module. Clauses get_output/10 were modified so that if such a transition is found the
output statement is replaced by the transition block. Transition combination implies that any reference to the input parameter of the combinee NFT will be replaced by the corresponding expression in one of the output statements of the combiner NFT. This symbolic replacement is performed by clauses repNm/4 explained in section 3.3.

Modifications of the program can be done easily as long as the structure of the input syntax tree does not change. Since the program is structured as the algorithm in section 5.2, further processing of an Estelle construct can be achieved by simply adding more goals to the appropriate place.

5.4. Performance of Module Merging

The performance experiments were executed with three specifications: a two module alternating bit protocol and a simple transport protocol adopted from references [3] and [2] and the finite state machine description of the transport protocol in [18]. The experiment was again run ten times and the execution times were averaged resulting in the table 5.1. Due to lack of real multi-module specifications the results have only indicative nature. The implementation of module merging required 1516 lines of Prolog code.

<table>
<thead>
<tr>
<th>input specification</th>
<th>size of input (lines)</th>
<th>size of output (lines)</th>
<th>runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>two module alternating bit protocol</td>
<td>266</td>
<td>227</td>
<td>4.812</td>
</tr>
<tr>
<td>normalized simple transport</td>
<td>2,071</td>
<td>4,248</td>
<td>18.428</td>
</tr>
<tr>
<td>transport (FSM only)</td>
<td>456</td>
<td>337</td>
<td>2.417</td>
</tr>
</tbody>
</table>

Table 5.1. Performance of module merging
CHAPTER 6

A TEST SEQUENCE GENERATION EXAMPLE

In this chapter we give an example of test design for the alternating bit protocol using the methodology introduced in chapter two. The specification of the alternating bit protocol used contains two modules: one describing the main behavior of the protocol; and a timer (appendix B). There are two types of PDUs exchanged with the peer process: DT, which carries data, and AK, which acknowledges the reception of data. The user of the protocol interacts with it using the primitives SEND_request, RECEIVE_request and RECEIVE_response. When the user wishes to establish communication he/she sends to the peer entity a SEND_request with a DT PDU assigned the sequence number 0. At the same time the user sends a TIMER_request primitive to a timer module and waits for an acknowledgement from the peer entity. The acknowledgement should arrive before the timer responds (primitive TIMER_response) otherwise the PDU sent is assumed lost and is retransmitted. This process is repeated until an acknowledgement is received within a certain time period after retransmission. Upon reception of an acknowledgement the sequence number of the next PDU to be sent becomes 1 (0) if the previous sequence number was 0 (1). During reception of data the PDUs received are acknowledged and stored in a buffer. The user can retrieve these PDUs by sending a RECEIVE_request primitive. Then PDUs are sent to the user carried by RECEIVE_response primitives. The communication of the alternating bit protocol with the lower layer is achieved via the DATA_request and LATA_response primitives.

First, normalization and the first phase of data flow analysis are applied.
The result is a normalized specification with all the PDUs identified. Next, module merging generates a single module specification (appendix C) which is passed through control and the second phase of data flow analysis. The output is used by a graphics tool to display the data and control flow graphs. The data flow graph is partitioned into protocol functions and subtours are generated from the control flow graph. The subtours covering each protocol function provide the test sequences. The graphics tools and subtour and test sequence generation programs were developed earlier and discussed in reference [23].

6.1. Normalization and First Phase of Data Flow Analysis

The data type defining the PDUs used by the alternating bit protocol is a variant record:

\[
\begin{align*}
\text{Id}_{-}\text{type} & = (DT, AK); \\
\text{Ndata}_{-}\text{type} & = \\
& \text{record} \\
& \quad \text{case Id: Id}_{-}\text{type of} \\
& \quad \quad DT: (\text{Data: Data}_{-}\text{type}); \\
& \quad \quad DT, AK: (\text{Seq: Seq}_{-}\text{type}); \\
& \quad \text{end};
\end{align*}
\]

We can submit this specification to the system discussed in the previous chapters using the command \textit{nf} (appendix A) which produces a normalized specification and identifies the PDUs carried by variables or interaction parameters. The arguments needed are the specification's file name and the name of the PDU record type.

6.2. Module Merging

The result of the procedure described in the previous section is processed by the \textit{cmbn} command with the following arguments: the normalized specification's file name, the name of the main module body (\textit{alt\_bit\_body}) as the name of the module where the merged machine will be placed, \textit{alt\_bit\_body} and \textit{timer\_body} as
the names of the modules to be merged, $[u, n]$ as the list of external interaction points and $[s]$ as the list of internal interaction points.

6.3. Control Flow and Second Phase of Data Flow Analysis

After module merging the command $df$ is applied. Its argument is the merged specification's file name. The result is a table describing the major state changes along with input and output interactions (control flow analysis) and a table of nodes and arcs describing the flow of data from the input to the output over the context variables (second phase of data flow analysis). This is the last step of processing done by the system described in this thesis.

6.4. Partitioning of Data Flow Graph to Protocol Functions

The data flow information generated by the previous step is fed into a graphics tool called $dfgtool$ which displays the data flow graph on a SUN workstation. Input interaction parameters are placed in the upper part of the data flow graph. Similarly output interaction parameters are placed in the lower part. The other nodes representing variables, constants, procedures and functions are placed in the middle. The numbers next to the arcs correspond to transition numbers.

The data flow graph consists of a number of blocks. Each block is automatically generated by the $dfgtool$ and represents the flow over a variable or related variables. These blocks can be merged interactively in order to describe protocol functions as mentioned in section 2.3. In our example four functions are identified: sending data, receiving data, sequencing the data or acknowledgement, and timing (i.e. statements that change control variables used by the timer). The user of $dfgtool$ can interactively merge the data flow graph blocks in order to generate functional blocks, each representing one of the four functions. Figure
6.1 gives the functions of the example protocol.

Figure 6.1. Functions of the alternating bit protocol
9.5. Test Generation

The last interactive tool used is called *testgen*. This tool initially produces subtours of the control flow graph of the specification (figure 6.2).

![Diagram](image)

Figure 6.2. Control flow graph of the merged alternating bit protocol

Each subtour corresponds to a sequence of transitions and starts from and ends at the initial state. The subtours must be checked considering the *provided* clauses. It is possible that the transition sequence defined by a subtour cannot occur if the *provided* clauses of one or more transitions are not satisfied. An interactive program called *editour* that does this job had been developed earlier, [23]. This program displays the transitions comprising a subtour and enables the user to change the subtours that cannot occur. In our example editing of the subtours was not necessary. Each of the protocol functions specified in the data flow graph partitioning is covered by a number of subtours. For example, function *sending data* is covered by the subtour

1 4 6 8 9 10 2

The input and output interactions of each transition in this subtour define a test sequence for this function. This test sequence is
estab_open u.send_req [n.data_req_dt] 1
ack_wait_open n.data_resp_dt [n.data_req_ak] 4
ack_wait_open n.data_resp_dt [n.data_req_ak] 6
ack_wait_open u.receive_req [u.receive_resp] 8
ack_wait_open null null 9
ack_wait_open null [n.data_req_dt] 10
ack_wait_open n.data_resp_ak null 2

Each line contains the from state, input, output and transition number of a transition in the subtour. The initial and final state is estab_open (the last transition, 2, drives the machine to estab_open upon reception of a data_resp_ak primitive from the lower layer). Each input in the above sequence puts the protocol to a new state and results in an output which can be observed by the tester. Any variation from this sequence is an indication of error. The subtours covering the rest of the functions and the complete set of test sequences are given in appendix D.
CHAPTER 7

CONCLUSIONS

A system for processing Estelle specifications in order to derive data and control flow information has been implemented. This system has been tested with realistic protocol specifications and we concluded that it can process real protocols efficiently. The current implementation is portable to any system that can run YACC, LEX and has a C and Quintus Prolog compiler. Quintus Prolog also offers a utility which can produce executable code running independently without invoking the Prolog environment.

The processing of Estelle specifications creates specifications whose behavior is equivalent to the original one. The changes implied by normalization and data flow analysis aim to the production of a simplified specification from which test sequences can be drawn easily. The specification resulting from the analysis is not meant to be implemented since features such as transition atomicity and channel definitions are not conserved.

The experience gained by processing large specifications leads to a major future modification that will increase performance and efficiency: the syntax tree can be created in fragments representing autonomous parts of Estelle code, i.e. declarations, initialization and transition bodies. Instead of processing the whole syntax tree it is possible to process each autonomous part separately, output the result and free the memory from the processed part. This modification may increase the execution time by increasing input/output. In the case of the system developed, much of the execution time is spent in recovering from stack overflows and new memory space allocation. If small syntax tree fragments are used and
the clauses use failure instead of recursion to implement iteration the memory overflows can be reduced or even eliminated. Another advantage of this modification is that it makes the system portable to smaller machines with higher memory size restrictions than a SUN workstation.

Another modification resulting in time and space efficiency is the reduction of the size of the names of the syntax tree nodes. This modification should be done when the development is entirely completed because otherwise, it may make the syntax tree unreadable by a human, thus creating problems in debugging.

Module merging was achieved using a limited reachability analysis algorithm. This algorithm can reveal errors in inter-module communication such as deadlocks, channel overflows and unspecified receptions. The power of this algorithm was not completely used. Extending the module merging program to perform limited reachability analysis and using it along with the programs for normalization and data and control flow analysis creates a protocol design tool. This tool can be used for validation of protocols against the previously mentioned errors.
REFERENCES


[22] B.Sarikaya, M.Barbeau, S.Eswara and V.Koukoulidis, "Improvements to the


A.1. Introduction

The system described in this guide processes a protocol specification in Estelle in order to derive control and data flow information which can be used by other tools to draw control and data flow graphs.

To follow this guide and be able to interact with the system, you must be familiar with the concepts of protocol analysis used by the system. Terms like lexical, syntactic and semantic analysis, normalization, data flow analysis, control flow analysis and module merging are defined in the Glossary (section A.4).

It is assumed that you have access to a Quintus Prolog compiler on your local computer system. Once you have the system installed you can enter the Prolog environment and load the programs needed for the analysis of a protocol by typing the command

```
norm
```

This loads to Prolog all the routines you need to use the commands described in section A.2.

A.2. Commands

Once the system is invoked it prints the familiar Prolog prompt

```
| ?- 
```

indicating that it is ready for your commands. If you wish to exit type the
command

* halt.

You must always end your command with a full-stop mark. Prolog considers newlines as character separators and reads the command typed until the first full-stop unless the full-stop mark is surrounded by single quotes.

After you type a command the system responds by printing information about its processing state and, possibly, error messages and/or warnings. The description of each command tells what information is displayed during execution.

A.2.1. "nf (Normal Form) Command

The nf command normalizes an Estelle specification. If you wish you can apply the first phase of data flow analysis by identifying the record types defining the PDUs. The syntax of this command is

\[ \text{nf('Estelle-specification')}. \]

for normalization only of the input Estelle specification, or

\[ \text{nf('Estelle-specification', PDU-record-type)}. \]

for normalization and the first phase of data flow analysis. The argument PDU-record-type must be given in lower case letters regardless how it appears in the input Estelle specification. The name of the output normalized specification is derived by suffixing the name of the input specification with .LIST.

A typical interaction with the system using the nf command has the following form
A.2.2. cmbn (Combine Modules) Command

If the specification contains more than one modules which should be merged, the you can invoke the cmbn command. The syntax of this command is

cmbn('normal_form_spec', parent_module, module1, module2,

    external_interaction_point_list, internal_interaction_point_list).

where normal_form_spec is the name of a normalized Estelle specification, parent_module is the body name of the parent module of the modules to be merged, module1 and module2 are the body names of the modules to be merged, and the last two arguments are the lists of the external and internal interaction points. External interaction points are used for communication of module1 and module2 with other modules and internal interaction points are used for communication between module1 and module2. The name of the output file is derived by suffixing the name of the input file with .MRG.

A typical execution of this command results in the following sequence

iv16. | ?- cmbn('example_spec.e.LIST', parent_body, module_body1, module_body2,
            [eilp1, eilp2, eilp3, eilp4], [eilp1, eilp2]). INPUT FILE: tp.e.LIST LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS

0 syntax errors, 0 warnings, 0 other errors detected
A.2.3. dtf (Data and Control Flow) Command

The *dtf* command is applied to the normal form specification generated by the *nf* or *cmbn* command. This command generates information that can be used by graphics tools for displaying the data and control flow graphs.

The syntax of *dtf* command is

```
dtf('normal_form_spec').
```

For each module body two files are generated: one containing data flow information and one containing control flow information. These files are named after the corresponding module body name suffixed with .DTF and .CTRL, respectively.

A typical interaction with *dtf* is shown below.

```
tab(\*); lv16. |?- dtf('example_spec.e'). INPUT FILE: example_spec.e.LIST LEXICAL, SYNTACTIC AND SEMANTIC ANALYSIS
0 syntax errors, 0 warnings, 0 other errors detected
```

A.3. Warnings and Error Messages

This section describes the error messages occurring during the analysis of a specification.
A.3.1. Compilation Phase Warnings and Error Messages

During the lexical, syntactic and semantic analysis phase -- preceding every command -- warnings, syntax and semantics errors are listed along with the line number on which they occurred.

Rename "$X$" if needed in the dictionary

Conflict with "$Y$":

An attempt to insert to the dictionary an identifier with definition "$X$" was made but this identifier was already inserted with a different definition "$Y$". For example this may happen if a constant is defined within an enumerated type. This warning does not occur for identifiers defined within the scope of different structures (e.g. fields of different records may have the same names). This message does not interrupt execution and the user may ignore it for identifiers related to scalar or subrange types.

A.3.2. nf Command Error Messages

line "$X$": "$Y$" is not a single statement function

If $Y$ is a function whose body cannot be reduced to a single statement with symbolic execution and $Y$ is called inside a provided clause of a transition, the above message is displayed. You should replace the function with another one consisting of a single statement (or reducible to a single statement).

line "$X$": "$Y$" replaced by its value
another one consisting of a single statement (or reducible to a single statement).

line "X": "Y" replaced by its value

This is a warning. The user is notified that the variable Y used inside the condition of an if statement is replaced by its symbolic value. This is necessary when this variable is assigned a value by statements preceding the if's condition.

line "X": failed to determine range values of "I" in FOR statement

A for statement is symbolically expanded for each possible value of the index variable I. It is assumed that I has been statically defined, otherwise it is not possible for the system to determine the values of I and the user is informed about this situation.

line X: failed to determine PDU kind referred to in interaction "Y"

line X: failed to determine PDU kind referred to by variable "Y"

The kind of PDU carried by a variable or exchanged at an output interaction could not be determined by the context. You should add one statement in the transition containing line X such that the tag-field of a PDU type variable or interaction parameter is assigned a constant corresponding to a PDU kind.

A.3.3. Unchecked Problems

Nested procedure or function declarations are not supported. If you do such declarations no error message will be printed and the nested declarations will be ignored. Normalization does not support nesting of transitions and repeat statements. In case of nested transitions and repeat statements no action is taken and the corresponding part of the input specification remains unchanged.
A.4. Glossary

**control flow analysis:**

The generation of the transition table of the finite state machine described by an Estelle specification.

**data flow analysis:**

Protocol data unit (PDU) identification and generation of information modeling the manipulation performed on input interaction parameters or context variables in order to determine the values of output interaction parameters.

**dictionary:**

A data structure built as a sorted tree which is used for storing variable and data type declarations.

**lexical analysis:** The conversion of the input specification to a sequence of tokens to be submitted for syntactic analysis.

**module merging:**

The merging of two communicating modules having the same parent module in one module.

**normalization:**

The removal of constructs that introduce paths or transfer of control during the execution of a transition.

**semantic analysis:**

The process of collecting type information and verifying that operators and operands are used consistently within expressions and statements.

**syntactic analysis:**
The process of deciding if a sequence of tokens can be generated by a grammar.

**syntax tree:**

A hierarchical data structure representing the grammatical phrases of the input specification.
APPENDIX B

THE ALTERNATING BIT PROTOCOL

specification example system process;
{ The alternating bit protocol }

timescale seconds;

type
  Data_type = ...;
  Seq_type = integer;
  Id_type = (DT, AK);
  Ndata_type =
    record
      case Id: Id_type of
        DT: (Data: Data_type);
        DT, AK: (Seq: Seq_type);
      end;

{ Channel definitions }
channel U_access_point( User, Provider );
  by User:
    SEND_req(Udata: Data_type);
    RECEIVE_req;
  by Provider:
    RECEIVE_resn(Udata: Data_type);

channel N_access_point( User, Provider );
  by User:
    DATA_req(Ndata: Ndata_type);
  by Provider:
    DATA_resn(Ndata: Ndata_type);

channel S_access_point( User, Provider );
  by User:
    TIMER_req;
  by Provider:
    TIMER_resn;

{ Module header definitions }
module Alt_bit_type process;
  Ip
    U: U_access_point( Provider ) common queue;
N: N_access_point( User ) common queue;
S: S_access_point( User ) individual queue;
end;

module Timer_type process;
  lp S: S_access_point( Provider ) individual queue;
end;

{ Module body definitions }
body Alt_bit_body for Alt_bit_type;

type
  Buffer_type = ...;

const
  Empty = any Buffer_type; { empty buffer }

var
  Send_seq, Recv_seq: Seq_type;
  Send_buffer, Recv_buffer: Buffer_type;
  B: Ndata_type;

state
  ACK_WAIT, ESTAB;

stateset
  EITHER = [ACK_WAIT, ESTAB];

procedure Format_data(s: seq_type; d: data_type; var A: Ndata_type);
begin
  A.Id := DT;
  A.Data := d;
  A(Seq := s
end;

procedure Format_ack(s: seq_type; var A: Ndata_type);
begin
  A.Id := AK;
  A.Seq := s
end;

procedure Store(var Buf: Buffer_type; data: data_type);
primitive;

procedure Remove(var Buf: Buffer_type);
primitive;
function Retrieve(Buf: Buffer_type): data_type;
primitive;

function Buffer_empty(Buf: Buffer_type): Boolean;
primitive;

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

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

Initialize
to ESTAB
begin
    Send_seq := 0;
    Recv_seq := 0;
    Send_buffer := Empty;
    Recv_buffer := Empty;
end;

{ Transitions }
trans

{ Sending data }
when U.SEND_req
from ESTAB
to ACK_WAIT
begin
    Store( Send_buffer, Udata );
    Format_data(Send_seq, Udata, B);
    output N.DATA_req(B);
    output S.TIMER_req
end;

when S.TIMER_resp
from ACK_WAIT
to ACK_WAIT
begin
    Format_data(Send_seq, Retrieve(Send_buffer), B);
    output N.DATA_req(B);
    output S.TIMER_req
end;
end;

when N.DATA_resp
provided (Ndata.Id = AK) and (Ndata.Seq = Send_seq)
from ACK_WAIT
to ESTAB
begin
  Remove(Send_buffer);
  Inc_send_seq
end;

{ Receiving data }
when N.DATA_resp
provided Ndata.Id = DT
begin
  Format_ack(Ndata.seq, B);
  output N.DATA_req(B);
  if Ndata.Seq = Recv_seq then
    begin
      Store(Recv_buffer, Ndata.data);
      Inc_recv_seq
    end
  end;

when U.RECEIVE_req
provided not Buffer_empty(Recv_buffer)
from EITHER
to same
begin
  output U.RECEIVE_resp(Retrieve(Recv_buffer));
  Remove(Recv_buffer)
end;

end;  { End of the Alt_bit_body }

body Timer_body for Timer_type;

const
  Retran_time = any Integer;  { Retransmission time }

var
  Stop, Stop_is : boolean;

state
  OPEN;
Initialize
to OPEN
begin
    Stop := true;
    Stop_bls := true;
end;

trans
when S.TIMER Req
begin
    { Cancel previous timer }
    Stop := true;
    Stop_bls := false;
end;

trans
provided not Stop_bls
begin
    Stop_bls := true;
    Stop := false;
end;

trans
provided not Stop
delay ( Retran_time, Retran_time )
begin
    Stop := true;
    output S.TIMER Resp
end;

end;
end. { End of specification }
APPENDIX C

NORMALIZED AND MERGED ALTERNATING BIT PROTOCOL

```
specification example system activity;
timescale seconds;

type
data_type ...;
seq_type = integer;
id_type = (dt, ak);
ndata_type_dt =
  record
data: data_type;
  seq: seq_type
end;
ndata_type_ak =
  record
  seq: seq_type
end;
ndata_type =
  record
  id: id_type;
  data: data_type;
  seq: integer
end;

channel u_access_point(user, provider);
  by user:
    send_req(u_data: data_type);
    receive_req;
  by provider:
    receive_resp(u_data: data_type);

channel n_access_point(user, provider);
  by user:
    data_req_ak(n_data: ndata_type_ak);
    data_req_dt(n_data: ndata_type_dt);
  by provider:
    data_resp_ak(n_data: ndata_type_ak);
    data_resp_dt(n_data: ndata_type_dt);

channel s_access_point(user, provider);
  by user:
```
timer_req;
by provider;
timer_resp;

module alt_bit_type_timer_type activity;
lp
  u: u_access_point(provider) common queue;
  n: n_access_point(user) common queue;
end;

body alt_bit_body_timer_body for alt_bit_type_timer_type;

type
  buffer_type == ...;

const
  empty = any buffer_type;

var
  b_dt: ndata_type_dt;
  b_ak: ndata_type_ak;
  send_buffer, recv_buffer: buffer_type;
  send_seq, recv_seq: seq_type;

state
  ack_walt_open, estab_open;

procedure store(var buf: buffer_type; data: data_type);
primitive;

procedure remove(var buf: buffer_type);
primitive;

function retrieve(buf: buffer_type): data_type;
primitive;

function buffer_empty(buf: buffer_type): boolean;
primitive;

const
  retrans_time = any integer;

var
  stop, stop_bis: boolean;

Initialize
to estab_open
begin
  send_seq := 0;
  recv_seq := 0;
  send_buffer := empty;
  recv_buffer := empty;
  stop := true;
  stop_bls := true
end;

trans
{ 1 }
when u.send_req
from estab_open
to ack_wait_open
begin
  store(send_buffer, udata);
  b_dt.data := udata;
  b_dt.seq := send_seq;
  output n.data_req_dt(b_dt);
  stop := true;
  stop_bls := false
end;

trans
{ 2 }
when n.data_resp_ak
provided (true) and (ndata.seq = send_seq)
from ack_wait_open
to estab_open
begin
  remove(send_buffer);
  send_seq := (send_seq + 1) mod 2
end;

trans
{ 3 }
when n.data_resp_dt
provided (true) and (not (ndata.seq = recv_seq))
from estab_open
to estab_open
begin
  b_ak.seq := ndata.seq;
  output n.data_req_ak(b_ak)
end;
trans
{ 4 }
when n.data_resp_dt
provided (true) and (not (ndata.seq == recv_seq))
from ack_wait_open
to ack_wait_open
begin
  b_ak.seq := ndata.seq;
  output n.data_req_ak(b_ak)
end;

trans
{ 5 }
when n.data_resp_dt
provided (true) and (ndata.seq == recv_seq)
from estab_open
to estab_open
begin
  b_ak.seq := ndata.seq;
  output n.data_req_ak(b_ak);
  store(recv_buffer, ndata.data);
  recv_seq := (recv_seq + 1) mod 2
end;

trans
{ 6 }
when n.data_resp_dt
provided (true) and (ndata.seq == recv_seq)
from ack_wait_open
to ack_wait_open
begin
  b_ak.seq := ndata.seq;
  output n.data_req_ak(b_ak);
  store(recv_buffer, ndata.data);
  recv_seq := (recv_seq + 1) mod 2
end;

trans
{ 7 }
when u.receive_req
provided not buffer_empty(recv_buffer)
from estab_open
to estab_open
begin
  output u.receive_resp(retrieve(recv_buffer));
  remove(recv_buffer)
end;

trans
{ 8 }
when u.receive_req
provided not buffer_empty(recv_buffer)
from ack_wait_open
to ack_wait_open
begin
  output u.receive_resp(retrieve(recv_buffer));
  remove(recv_buffer)
end;

trans
{ 0 }
provided not stop_bls
from ack_wait_open
to ack_wait_open
begin
  stop_bls := true;
  stop := false
end;

trans
{ 10 }
provided not stop
delay(retran_time, retran_time)
from ack_wait_open
to ack_wait_open
begin
  stop := true;
  b_dt.data := retrieve(send_buffer);
  b_dt.seq := send_seq;
  output n.data_req_dt(b_dt);
  stop := true;
  stop_bls := false
end;

end;
end.
APPENDIX D

TEST SEQUENCES FOR THE ALTERNATING BIT PROTOCOL

Name of Function => sequencing

3 subtours

subtour : 0
estab_open
ack_wait_open n.data_resp_dt [n.data_req_ak] 4
ack_wait_open n.data_resp_dt [n.data_req_ak] 6
ack_wait_open u.receive_req [u.receive_resp] 8
ack_wait_open nil nil 9
ack_wait_open nil [n.data_req_dt] 10
ack_wait_open n.data_resp_ak nil 2

subtour : 1
estab_open n.data_resp_dt [n.data_req_ak] 3

subtour : 2
estab_open n.data_resp_dt [n.data_req_ak] 5

Name of Function => sending data

1 subtours

subtour : 0
estab_open u.send_req [n.data_req_dt] 1
ack_wait_open n.data_resp_dt [n.data_req_ak] 4
ack_wait_open n.data_resp_dt [n.data_req_ak] 6
ack_wait_open u.receive_req [u.receive_resp] 8
ack_wait_open nil nil 9
ack_wait_open nil [n.data_req_dt] 10
ack_wait_open n.data_resp_ak nil 2

Name of Function => receiving data
3 subtours

subtour : 0
estab_open u.send_req [n.data_req_dt] 1
ack_wait_open n.data_resp_dt [n.data_req_ak] 4
ack_wait_open n.data_resp_dt [n.data_req_ak] 6
ack_wait_open u.receive_req [u.receive_resp] 8
ack_wait_open nil nil 9
ack_wait_open nil [n.data_req_dt] 10
ack_wait_open n.data_resp_ak nil 2

subtour : 1
estab_open n.data_resp_dt [n.data_req_ak] 5

subtour : 2
estab_open u.receive_req [u.receive_resp] 7

Name of Function => timer

1 subtours

subtour : 0
estab_open u.send_req [n.data_req_dt] 1
ack_wait_open n.data_resp_dt [n.data_req_ak] 4
ack_wait_open n.data_resp_dt [n.data_req_ak] 6
ack_wait_open u.receive_req [u.receive_resp] 8
ack_wait_open nil nil 9
ack_wait_open nil [n.data_req_dt] 10
ack_wait_open n.data_resp_ak nil 2

Subtours not covered

Subtour 0
estab_open u.send_req [n.data_req_dt] 1
ack_wait_open n.data_resp_ak nil 2