Estelle-based test generation tool

Behçet Sarikaya, Behdad Forghani* and Srinivas Eswara* present a test design tool, using an extended FSM model, for functional analysis and test derivation of protocols

A test design tool for functional analysis and test derivation of protocols formally specified using an extended finite-state machine model is presented. The formal description language supported is Estelle. The tool's main components include a compiler, a normalizer, a multiple module transition tour generator and several interactive programs. The tool is based on a static analysis of Estelle called normalization, which is explained in detail with various examples. The normalized specification facilitates graphical displays of the control and data flow in the specification by the interactive tools. Next discussed is test generation, which is based on verifying the control and data flow. First the data flow graph must be decomposed into blocks where each block represents the data flow in a protocol function. From the control graph the tool generates transition tours, and then test sequences are derived from the transition tour to test each function. The performance of the tool on various applications is also included.

Keywords: specification languages, conformance testing, functional testing, data flow graphs

Distribution system specification and analysis have recently been an area of active research and development. An area of distributed system research that is of interest to us in this paper is communication protocols (providing reliable communication in distributed systems).

To provide reliable communication among computers from different manufacturers the Open Systems Interconnection (OSI) reference model has been defined to structure protocols in various layers. What makes OSI work is the standardization of its protocols and services. Standard definitions are presently given in natural language, and because of this the definitions contain ambiguities and are imprecise. Standardization institutions such as ISO and the CCITT have defined formal languages with which protocols and services can be specified. The language which is of interest to us in this paper is Estelle[1]. Estelle is based on an extended finite-state machine model.

Protocols and services of OSI are complex. They have to be thoroughly tested before being incorporated in the system. Another related activity is testing for conformance to the protocol specification. Conformance testing is usually undertaken by national or international institutions. There are two major aspects of conformance testing: defining the environment in which an implementation under test (IUT) will be tested; and defining the tests to be applied to the IUT[2].

Automated test design for protocols can be done using an FSM model and deriving test sequences from this model. However, FSMs lead to state-explosion when one tries to model important features of protocols such as interaction parameters and certain features of data transfer such as sequence numbers.

The computer-aided test design tool, nick-named Contest-Estl, is an implementation of a functional formal specification (Estelle)-based test design methodology[3].

Department of Computer and Information Sciences, Bilkent University, Bikent, Ankara 06533, Turkey
*Department of Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve West, Montreal, Quebec, Canada H3G 1M8
Paper received: 6 December 1990. Revised paper received: 23 February 1991

0140-3664/91/009534-11 © 1991 Butterworth-Heinemann Ltd

While the tests obtained from FSM-based tools can only have a limited control flow coverage, our tool provides facilities to derive tests to fully cover the control and data flow. We introduce Estelle below, along with the structure of the tool. The compiler and the normalizer are discussed, and the interactive tools examined. Performance of the tool on various applications is also given.

EXTENDED FINITE-STATE MACHINE MODEL

EFSM model describes the system (protocol) as a collection of modules. Each module is a finite-state machine capable of having memory, i.e. extended finite-state machine. Modules of an entity can communicate with each other as well as with the environment over channels (FIFO queues). Service primitives (exchanged with bottom and upper layer entities) and internal interactions (with other modules) are communicated in the channels. Protocol data units (PDUs), i.e. messages exchanged between two protocol entities, need not be explicitly defined, but are encoded and introduced as parameters to the service primitives. Decomposition of an entity into modules is usually functional: a module for timer management; a mapping module to map the PDUs into interactions with the environment, i.e. abstract service primitives (ASP); an abstract protocol module for handling service primitives and forming PDUs, etc.

The language of Estelle is based on Pascal with extensions to facilitate protocol specification. To save space we only describe the constructs related to transitions. FROM/TO clauses define initial/final state(s) of the finite-state machine, respectively. The arrival of an input interaction is expressed using WHEN. Transitions with no WHEN clause are called *spontaneous*. The conditions for firing the transition are described in a PROVIDED clause, which is a Boolean expression on interaction primitive parameters and variables of the module. Variables of the module are called *context* variables. Finally, the action of the transition is contained in a BEGIN block which can have assignments to context variables, calls to internal procedures, Pascal conditional statements and produce output with the OUTPUT statement.

Estelle supports nondeterminism (internal decisions of the entity, for example) by way of spontaneous transitions, and by allowing more than one transition from a given major state to have their predicates enabled. Once a transition is enabled, its execution is atomic. Abstractness of the specification, i.e. being away from implementation considerations, could be achieved through the use of incomplete type definitions using the three-dot notation, such as in:

buffer_type = ...;

and the use of abstract data types (buffer_type above is an abstract data type) with their operations defined as primitive procedures and functions.

It is possible to semi-automatically generate executable code from Estelle specifications. To this end, various translators (so called compilers) of Estelle have been developed[4, 5]. Semantics of Estelle is discussed in Courtiat 89[6], validation tools based on Estelle have been developed[7], and test design for different phases of protocols is discussed in Boyce and Probert[8].

The specification in Estelle of a simplified transport protocol, hereafter called TP2, will be used as an example in the paper. This specification describes the transport entity composed of two modules: the Abstract Protocol Module (called ap_body) which handles protocol functions and interacts with the users; and the Mapping Module (called map_body), which handles the mapping of the transport PDUs into network service data units, and *vice versa*. The ap_body module body contains 20 transitions and the map_body has four transitions.

Contest-Estl accepts all of the constructs of Estelle. It extends the subset of Estelle that is covered by the test design methodology[3] to handle modular specifications, nondeterminism and all Pascal constructs except pointers. No specific action is taken for Estelle features such as dynamic module creation/destruction, (system) process/activity, and transition priorities in the present version.

TOOL STRUCTURE

Contest-Estl takes an Estelle specification and semi-automatically produces tests for the input specification. Figure 1 shows the global structure of the tool. Each of the three components are explained in detail in the following sections, while this section gives an overview of each.

After compilation the normalizer is activated. The process of normalization transforms the input specification into another (Estelle) specification which (possibly) contains more transitions, each having single paths. Such a specification is called a *normalized specification*. In a normalized specification we identify two types of flows: *control flow*, which models major state changes from one transition to another; and *data flow*, which models flow from input primitive parameters (service primitives and/or PDUs) to context variables, and from context variables to the output primitive parameters. The normalizer generates intermediary forms for these graphs which are passed to the interactive tools.

The interactive tools display the control and data flow graphs, and finally generate test sequences. A representation of various protocol functions can be interactively obtained from the data flow graph. The transition subtours obtained from the control graph yield the sequences of interactions to effectively test each function. Most of these functions can be tested independent of each other with the application of the transition subtours.

COMPILER

To ensure a correct specification for the normalizer, the input specification is syntactically and semantically

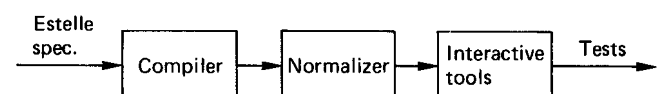

Figure 1. Global structure of Contest-Estl

analysed. There are virtually four phases of processing the input Estelle specification: lexical analysis, syntactic analysis, semantic analysis and global parse tree and symbol table construction. In the actual implementation we combine all these phases together into a single pass through the Estelle source code. Our compiler module is implemented using the standard tools Lex and Yacc available in the Unix operating system, and its structure is inspired from the NIST Estelle compiler[5].

Since normalization is implemented in Prolog, we convert the parse tree to a Prolog clause using a conversion program written in C. This program converts the symbol table into a sorted tree structure called a dictionary[9]. A dictionary is defined recursively:

```
dic(<name>,<value>,<dic-1>,<dic-2>)
```

pairs <name> with <value>, where <dic-1> and <dic-2> are subdictionaries. The dictionary is ordered with respect to <name>.

Normalizer

The normalizer works in two steps: first, a kind of symbolic execution is applied to the input specification to transform it to a normalized specification. The normalized specification can be inspected and modified by the user. Next, we use the compiler module again to do syntactic and semantic checks on this normalized specification, and extract the control and data flow information from it to be used by the interactive tools.

Symbolic execution is a technique for static program analysis[10]. It has been used for program verification and testing. An Estelle specification can be symbolically executed for the purpose of identifying all the control paths. It is also possible to express these paths as distinct transitions using the Estelle syntax.

In Contest-Estl we use Prolog for normalizing Estelle specifications. Prolog has been used for implementing language processors including compilers in a compact way (possibly less efficient in terms of execution time)[11]. Other uses of Prolog in protocol analysis and testing are discussed in Sidhu[12] and Ural and Probert[13].

Normalization is done in various steps. Basic normalization, as described elsewhere[3], is implemented as follows: body replacements are done for local procedure and function calls; conditional statements (IF, CASE) are eliminated; FROM and TO clauses are processed so that each transition contains at most a single major state change. Provided clauses are put into disjunctive normal form with the elimination of ORs. Extensions to the basic normalization include the following: declarations are processed to simplify the complex data structures such as variant records; WITH statements are removed by record structure replacements. Each module of the specification is treated independently, resulting in a normalized specification for each module. Steps of normalization are detailed in what follows.

Processing the declarations

Variant records are converted into records enumerating all case constants. For example, the following variant record defining the PDUs of the alternating bit protocol:

```
Id_type = (DT,AK);
  Ndata_type =
        record
           Conn: Cep_type;
           case Id: Id_type of
           DT: (Data: U_Data_type);
           ACK: (Seq: Seq_type);
        end;
gets expanded to:
     ndata_type_dt =
           record
              conn: cep_type;
              data: u_data_type
           end;
        ndata_type_ack =
           record
              conn: cep_type;
              seq: seq_type
        end;
```

Procedure/function call elimination

Each procedure/function body definition is converted to an internal representation to facilitate replacement procedure. Local variables are converted to global variables with unique identifiers, and a global variable is sometimes created for each parameter which is called by value, i.e. when the parameter is assigned a value. For the result returned by each function, a global variable is created.

All procedure/function calls are replaced by the corresponding begin-end block. In replacing a procedure call, here is what happens:

- The begin-end block is obtained from the dictionary.
- For each value parameter which is assigned in the body of the procedure, an assignment statement is placed before call replacement occurs.
- All other parameters are symbolically replaced by the actual parameters.

As an example, the procedure definition:

```
procedure P (v,x:integer; var y:integer);
var z:integer;
begin
      z: = x;
      x: = v + z + y;
end;
```

with the following call:

```
P(1, 2, k)
```

produces the code:

```
P_x: = 2;
P_z: = P_x;
P_x: = 1 + P_z + k;
```

Function call replacement is similar, and is explained in Barbeau and Sarikaya[14]. Recursion is not handled in the present version of the tool.

Conditional statement elimination

Conditional statements in the transitions can be eliminated by creating two or more equivalent transitions and modifying the PROVIDED clause to reflect the condition for taking this path. As an example, the transition of the example protocol:

```
WHEN Map.transfer
PROVIDED PDU.kind = DT
FROM open TO same
  begin
    if (RCr <> 0) and (PDU.sendseq = TRseq) then
    begin
        TRseq: = TRseq + 1;
        RCr: = RCr − 1;
        OUTPUT TS.TDATAind(PDU.user__data,
        PDU.end__of__TSDU)
    end
    else error;
  end;
```

produces the following normalized transitions:

```
trans
{015}
when Map.transfer
provided (PDU.kind = DT) and (RCr <> 0) and
(PDU.sendseq = TRseq)
from open
to open
begin
  TRseq: = TRseq + 1;
  RCr: = RCr − 1;
  OUTPUT TS.TDATAind(PDU.user__data,
  PDU.end__of__TSDU)
  end;

trans
{016}
when Map.transfer
provided (PDU.kind = DT) and not ((RCr <> 0)
and (PDU.sendseq = TRseq))
from open
to open
begin
  error {a primitive procedure}
end;
```

Note that the normalized transitions are sequentially numbered for each module, and all the transitions in the second module (the AP module in the example protocol) contain a 0 (zero) in the front, the third module a 00, and so on.

Modification of the PROVIDED clause is complicated in cases where the Boolean expression of the conditional statement contains variables assigned in the same transition before the conditional statement. Normalizer's symbolic replacement feature is invoked in these cases. The symbolic value of the variable is computed by symbolic execution, and this value replaces the variable in the Boolean expression of the conditional statement[15].

Elimination of the CASE statement is a generalization of the IF statement, i.e. several transitions are created corresponding to each arm of the case statement. Loop statements (for and while) are eliminated by repeating the body of the loop for every index variable value. In cases where exhaustive enumeration is not possible, a limited number (usually three) executions of the loop body is considered.

FROM/TO clauses

This step simplifies FROM/TO clauses in a given specification. State sets are eliminated by creating more than one transition, one for each state in the set, such as in (taken from alternating bit protocol):

```
from EITHER
to same
  when U.RECEIVE__request
  provided not buffer__empty(Recv__buffer)
  begin
    Q.Msgdata: = Retrieve(Recv__buffer);
    output U.RECEIVE__response(Q.Msgdata);
    Remove(Recv__buffer)
  end;
```

where EITHER is a state set containing the states ACK__WAIT and ESTAB. This transition transforms into:

```
trans
  {2}
  when u.receive__request
  provided not buffer__empty(recv__buffer)
  from estab
  to estab
  begin
    {same as above}
  end;

trans
  {3}
  when u.receive__request
  provided not buffer__empty(recv__buffer)
  from ack__wait
  to ack__wait
  begin
    {same as above}
  end;
```

WITH statements removal

The record variable access in a WITH structure is appended to the beginning of each variable access inside WITH's scope, provided that this variable is a field of the type of the record variable access. For example, this WITH statement block:

```
with B do
  begin
    Id      : = ACK;
    empty (Data); {no data for an ACK}
    Seq     : = Q.Msgseq;
  end;
```

is changed to:

```
empty(b__ack.data);
b__ack.seq: = q.msgseq;
```

PDU field identification

The first phase of the normalizer produces a normalized specification which should be submitted to the second phase. The second phase has the aim of identifying the PDU fields and making the PDU processing explicit in both the channel definitions and transitions. Identification of individual fields of all the input/output interactions is necessary for the data flow graphs. For ASPs, Estelle specifications contain an explicit list of fields, but such is not the case of the PDUs. It is common practice to define a single variant record for PDUs and identify them using an identification field. In the example protocol specification *TPDUandCtrlInf* is used to define the fields of all the PDUs (DT, CR, CC, etc.), each identified by the field *id__type*. The name of the variant record that defines the PDU fields is provided by the user.

The normalizer modifies the channel definitions to explicitly list the PDUs exchanged in interactions happening on the channels. The channel definition modifications for the example protocol follows. The external interaction point *nceprims* gets redefined as:

```
channel nceprims(user, provider);
  by user:
    ndatareq__cc(nsdufragm:tpduandctrlinf__cc;
      lastnsdufragm:boolean);
    ndatareq__cr(nsdufragm:tpduandctrlinf__cc;
      lastnsdufragm:boolean);
        ...{other PDUs, such as DR, DT, etc.}
  by provider:
    ndatareq__cc(nsdufragm:tpduandctrlinf__cc;
      lastnsdufragm:boolean);
    ndatareq__cr(nsdufragm:tpduandctrlinf__cc;
      lastnsdufragm:boolean);
        ...{other PDUs, such as DR, DT, etc.}
```

The internal interaction point *pduandctrlprims* becomes:

```
channel pduandctrlprims (protocol mapping);
  by mapping:
    transfer__cc(pdu:tpductrlinf__cc);
    ...
  by protocol:
    transfer__cc(pdu:tpductrlinf__cc);
    ...
```

After having processed the declaration part, the transitions have to be modified so that input and output interactions and assignment statements will explicitly indicate the PDUs being exchanged or modified. First the PROVIDED clause of the transition is scanned to find out if the interaction name in the WHEN clause refers to one of the PDUs. On the affirmative, the interaction name is changed to the corresponding name in the channel definition. The transition block is scanned next. Any references to the PDU variant record are replaced in accordance with the variant record's enumeration. OUTPUT statements are modified to reflect the specific PDU which is the output.

Second phase of normalization

The second phase of the normalizer generates as outputs

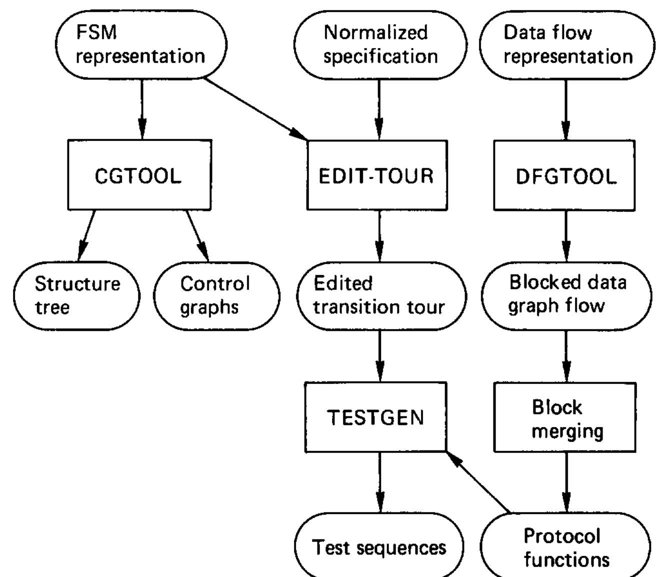

Figure 2. Interactive tools global structure

a normalized Estelle specification in printable format, and three types of intermediary files explained below. These files become input to the interactive tools component.

The normalizer extracts the name of the main module, as well as any submodules, and creates an intermediary file (with the .struct suffix) to become the static structure representation in the form of a tree.

For each module, the normalizer extracts the names of the inputs (WHEN clause), present states (FROM clause), next states (TO clause) and outputs (OUTPUT clause), and creates an intermediary file (.ctrl suffix) to become the FSM representation (see Figure 2).

The input to the data flow generation tool is a description of the action part of the transitions in the form of a node list and an arc list (.dtf suffix), also on a per module basis.

Arrays and sets

Some processing is required to represent each array (set) used in the specification with a single node in the data flow graph. Array/set references in the transitions must be eliminated and replaced by procedure and function calls for assigning to an array and for accessing an array element, respectively. An assignment statement of the type:

a[i]: = b; is converted to a procedure call:
assign__array(a,i,b);

and an assignment of the type:

b: = a[i];

is converted to a function call:

b: = index__array(b,i);

Similar processing is done for set operations in the transitions. This processing changes only the input to the data flow graph generation tool (.dat files), and the source text of the normalized specification remains unchanged[15].

Simplification

In this step PROVIDED clauses of the normalized transitions are processed. Simplification looks for predicates such as:

PROVIDED a or b
{the rest of the transition}

and generates three transitions (from the truth table of logical or) to replace the one above:

PROVIDED a and b {the rest of the transition}
PROVIDED not a and b {the rest of the transition}
PROVIDED a and not b {the rest of the transition}

This step facilitates automatic test data generation, since each PROVIDED clause can be satisfied by assigning a single value to each parameter of the input primitive (PDU or ASP)[16].

After normalization we obtain 49 transitions for the Map and 37 for the AP modules of the example protocol. After simplification, the total number of transitions increased to 88, i.e. only two more transitions were added.

INTERACTIVE TOOLS

This section explains three tools: cgtool, dfgtool and testgen; cgtool to display the structure and control graphs, dfgtool to display the data flow graph, and then testgen to generate test sequences. The structure of the interactive tools is shown in Figure 2, which describes cgtool, testgen and dfgtool from left to right.

Cgtool

Static modular decomposition of an Estelle specification can be visualized in the form of a tree, called a structure tree.

Major state changes from one normalized transition to another is called control flow. Control flow is best shown by a state diagram.

Cgtool first displays the structure tree based on the .struct file. Figure 3 shows the structure tree of the example specification called simple__tp. It has a tp__body module which is decomposed into two modules, ap__body and map__body. By clicking the mouse on the nodes corresponding to modules the user can visualize the control flow graphs which are displayed using .ctrl files.

Cgtool is implemented in C. The graphs (structure tree and FSM) are displayed by cgtool using Sun workstation graphics[17]. The cgtool displays the machine by first laying out the first node (specification name of the .struct file or the initial state of the .ctrl file) and then the transitions from the first node along with their next states in the graphics workspace. Self-loop transitions are shown as small circles around the state. The initial state is distinguished from the others by a special representation. On

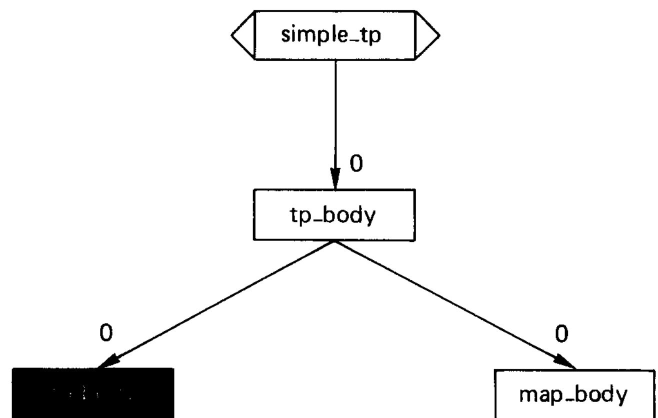

Figure 3. Structure tree of simple__tp

each arc shown are the label(s) of the normal form transition(s) (except for .struct, where there are no transition numbers).

After displaying the control graph of the specification, cgtool becomes a menu-driven interactive tool to let the user move the states left, right, up and down as well as compress/expand it horizontally/vertically by means of the mouse. The graph can be saved anytime by a save command in the menu. In this case, cgtool saves the coordinates of the states, the control points of the curves (outgoing and self-loop transitions). The *info* command in the menu is for obtaining more information about the transitions. The input and output interactions of any normalized transition can be obtained by typing the transition number at the subwindow created by the info command. The FSM for ap__body module of transport protocol is displayed in Figure 4.

Dfgtool

Actions of the normalized transitions can be seen as a collection of operations which process parameters of the

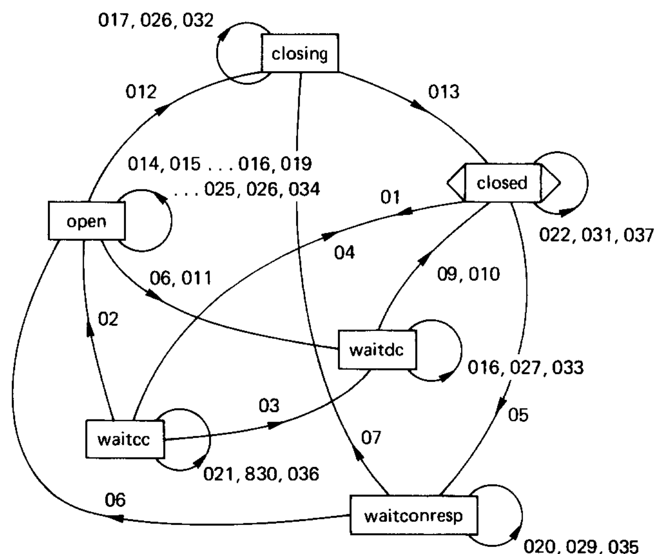

Figure 4. Control graph for the ap__body

input interactions (ASPs, internal interactions and/or PDUs) in order to determine the values of parameters of the output interactions. This processing is done using context variables as storage and applying certain functions such as arithmetic operations or abstract data type operations on context variables. Thus we define value changes on the context variables as data flow. Data flows from the input interaction parameters to the context variables, and from context variables to the output interaction parameters. This leads to a natural graphical representation of the actions of normalized transitions, called a data flow graph (DFG). In the case of protocols, generation of DFGs is only possible when PDU (being input/output interactions) parameters are explicitly identified. This point was discussed in the PDU field identification section above.

Description of DFGs

In the upper part of the data flow graph, data sources (input interaction parameters) are placed, and similarly, data sinks (output interaction parameters) are placed in the bottom. All other nodes are placed in the middle. The arcs describe the information flow of the statements in the actions. For example, for a simple assignment statement, an arc is created from the source node (the variable on the right hand side of the := operator), to the destination node (the variable on the left hand side of the := operator).

Procedure/function parameters are represented depending on whether they are passed by value or by reference: Value parameters are connected by an arc to the node representing the procedure; reference parameters are connected to the node by a two directional arc, since the parameter can be at times an input or an output.

The arcs are labelled with the number of the normalized transition in which the statement it models can be found. A given transition number can be found on more than one arc since, in general, a transition contains more than one statement. The same assignment statement occuring in more than one transition is represented with a single arc containing a list of transition numbers. A given operation (procedure/function call) used in different places is separately represented one for each of its applications. Thus a given procedure identifier can appear more than once in the data flow graph. The enabling predicates of the transitions, i.e. the PROVIDED clauses are not explicitly represented in the DFG to avoid cluttering of the graph; these predicates are taken into account during test generation.

We define six types of nodes in a data flow graph: EI (EO) nodes representing input (output) ASP parameters; II (IO) nodes representing input (output) PDU and internal interaction (if any) parameters; D-nodes for the variables; and F-nodes for the functions. Input/output nodes are represented with the same icon, and the differentiation is made by position (top screen for I and bottom screen for O-nodes) and by reverse-videoing the node names of the internal nodes (II or IO).

A DFG from the example protocol is shown in Figure 5. The normalized transition 011 reads as:

```
any reason:reasontp do
provided reason <> ts_user_init
from open to waitdc
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;
     output map.transfer_dr(pdu_dr_pdu_dr)
end;
```

The first output statement and the following two assignment statements are shown by three different arcs in this figure.

DFG generation

The dfgtool is also implemented in C. The normalization phase generates an internal representation of the graph consisting of the nodes and the arcs in separate files for each module. Dfgtool uses Sun workstation graphics facilities to display the graph.

The dfgtool, like cgtool has a menu-driven interactive user interface. The graph can be scrolled to left (right) using the left (right) menu command; blocks can be displaced and the screen can be refreshed using the *move-block* and *redisplay* commands, respectively. Straight-line arcs can be replaced by curves using the *curve* menu button.

Partitioning the DFG and block merging

A data flow graph can be partitioned into blocks, a block representing the flow over a single context variable. The partitioning algorithm used in the tool creates a separate block from each D-node, and includes into this block all nodes that are linked (by incoming and outgoing arcs). Dfgtool, when called, displays the partitioned DFG of the module with vertical lines between each block.

Sarikaya[3] has shown that by merging some of the blocks, it is possible to obtain representations of most of the protocol functions (usually identified as data transfer for sending, data transfer for receiving and flow control for sending and receiving, etc.). Some of the rules for merging to be applied until no further merging can be accomplished are summarized.

Blocks Bi and Bj can be merged if their output nodes are of the same data type. If the types of the input nodes of Bi are also contained in the output nodes of Bj, the two blocks could be merged. Blocks Bi and Bj containing the data nodes that are related could be merged.

Merging can only be automated to some extent since what variables were used to specify a protocol function is a piece of information that cannot be extracted automatically. The dfgtool does not attempt automatic merging, but its menu contains a number of menu commands to facilitate interactive merging.

The *merge-help* command, when invoked, displays a menu with three choices: possible merges to give a list of blocks that can be merged; general information to display information to guide the user in doing further merges; and quit to return to the main menu. The possible merges list is obtained from the types of the input/output interaction

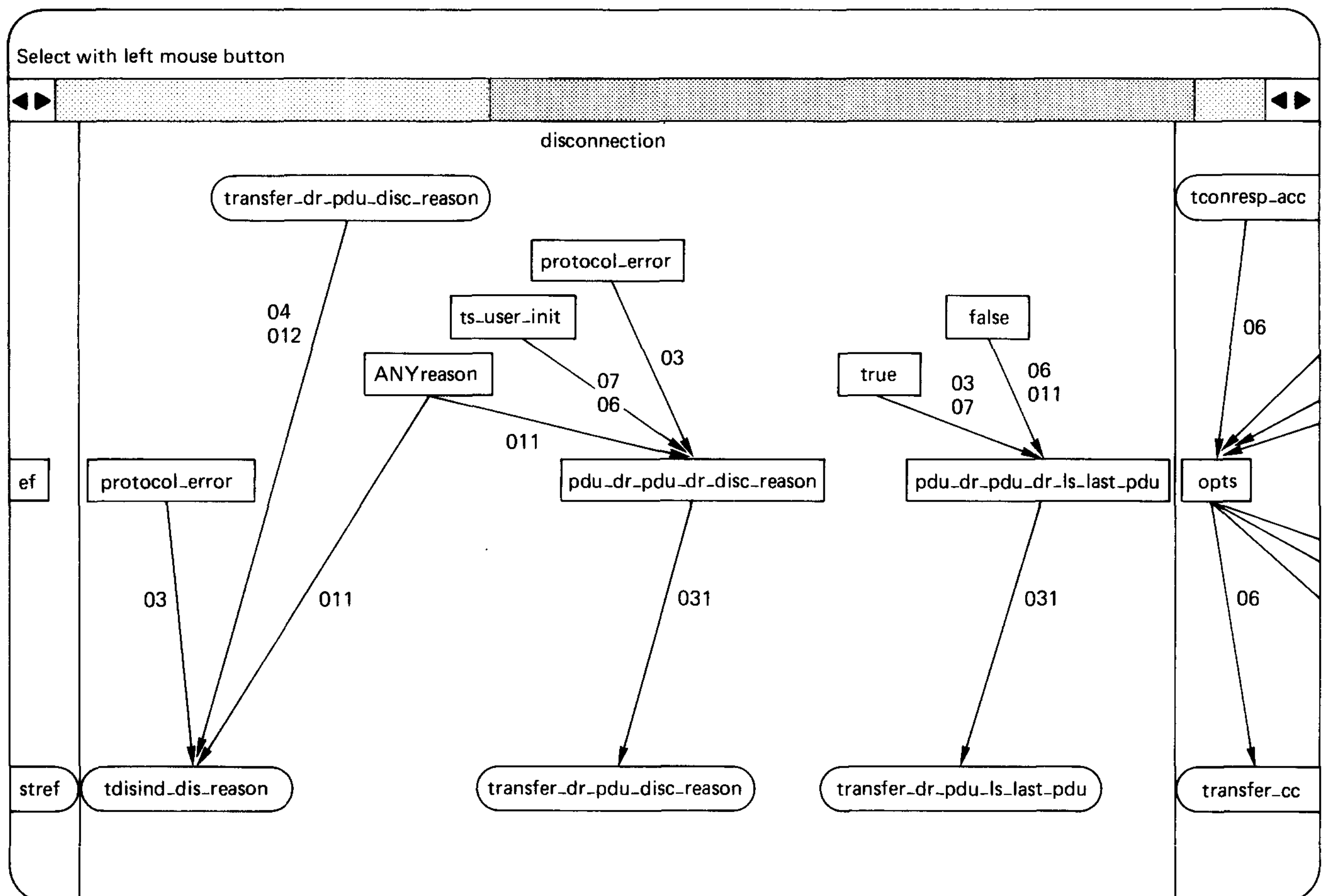

Figure 5. DFG of the ap__body module

parameters, i.e. blocks containing input or output interaction parameters which are of the same type can be merged. The *merge* command does the physical merging. The *name-block* command lets the user give names to the blocks to be later used as test purposes by the testgen tool. The names are displayed at the top of each block. Initially, dfgtool displays the blocks with default names (block1,... blockn). The *write* command saves the block names given and the list of merged blocks in that session. The *save* command also creates a file (.dat file) containing function names and all the distinct transition numbers of the arcs that exist in the function. Using just the quit command exits the tool without saving the changes.

Block merging process yields 11 functions for the example protocol: sender credit value, options, acknowledgements, disconnection (see Figure 5), etc., for the ap__body and disconnection, fragmentation, etc., for the map__body module. In most cases, the functions will be specific to the modules, while rarely a given function (disconnection, for example) will have some flow in more than one module.

Testgen

The last interactive component of Contest-Estl is the test sequence generation (testgen) tool. It is invoked after displaying the control graph and the data flow graph, and after block merging of all the modules is completed. Testgen generates sequences of interactions from the control graph(s), lets the user incorporate the effect of the enabling conditions (PROVIDED clause) of the normalized transitions into these sequences, and then generates test sequences to effectively cover the data flow in each function.

Testgen is implemented in C with an interface to the workstation graphics software to create menus and text subwindows. Its different components are explained in the sequel.

Subtours generation

To be able to handle modular specifications, we extend traditional definitions of transition tour and subtour[18, 19]. A depth-first coverage of the transitions of all the modules is called a *transition tour*. Each subsequence of a transition tour that starts and ends in the initial state(s) is called a *subtour*. The subtours have the interpretation that they result in the interaction sequences of the tests with distinct interpretation. They include a number of phases, and these phases give rise to a useful interpretation by the test designer such as subtours for data transfer tests, call refusal tests, etc.

Subtours generation component of testgen is called *mstourgen*. It takes the FSM representation(s) (.ctrl files) as

input, generates a transition tour and then divides it into subtours.

The main difficulty in generating a multiple module tour generation is in avoiding deadlocks. Transitions which take their input from internal channels cannot be directly executed. Before an output is sent to the internal channel, the other module must be checked to see if it is in a state that can accept the output. We also make sure that the resulting tests are synchronizable[18].

For the example protocol, the initial states of the modules are: idle for map__body and closed for ap__body. The first transition included in the tour is the transition 01 of ap__body:

closed ts.tconreq [map.transfer__cr] waitcc 01

which means that *ts* (transport user) sends tconreq, since ap__body is ready to accept it. Since *ts* sent the last ASP to ap__body, and map__body may consume tconreq immediately, both may next fire a transition. Then some spontaneous transitions are selected:

idle ap.transfer__cr nil idle 6 {idle is the only state
waitcc ts.u__ready nil waitcc 021 of map__body}
waitcc nil [ts.ready] waitcc 036
idle nil nil idle 9
...

At this point, transition 43 in map__body sends transfer__cc to ap__body and transition selection continues. Complete transition tour is not listed to save space.

A possible extension of mstourgen is in incorporating the effects of process/activity attribute of Estelle module bodies. The result would be to reveal possible parallelism in transition execution, e.g. process transitions are allowed to execute in parallel. Mstourgen could be modified to generate subtours in tree form instead of in linear list form by considering parent-child relationships of modules and processes and activities. It can be shown that only events (inputs and outputs) need be considered for such a modification.

Edit-tour

Subtours are a way of sequencing the normalized transitions based on the state sequence, without considering semantic information such as enabling conditions of the normalized transitions. Some of the subtours may be infeasible, i.e. this sequence of transitions cannot be executed due to conflicting enabling predicates. Automatic elimination of the *infeasible paths* is in general unsolvable[10]. Edit-tour is designed to help the user to interactively go through the subtours and then modify should an infeasible path be detected.

Edit-tour displays the transition numbers in one of the windows, the current subtour in another, and finally the text of the normalized transitions that occur in the subtour in a text subwindow that could be scrolled. The user can identify any conflicts in the PROVIDED clauses of any transition that follows each other. Edit-tour lets the user interactively update the tour. An example display from edit-tour is shown in Figure 6. As we see from the figure,

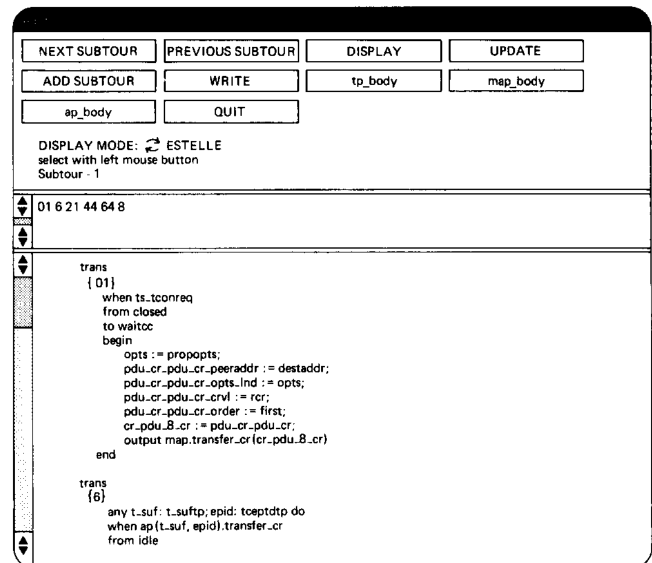

Figure 6. Test sequence display

edit-tour lets the user select the next/previous subtour generated by mstourgen, add a completely new subtour, and see the transitions of the main module (tp__body) and its submodules (map__body and ap__body). When the editing is completed, the user chooses the write button to save the results, or otherwise quits without saving.

Edit-tour can also be used in incorporating transition priorities into the subtours. The user can easily change the order of transitions in a subtour to give the effect of the priority clause.

Test sequence generation

Data flow functions obtained from dfgtool must be tested using the subtours obtained from the control graph. For this purpose, testgen uses the files (with .dat suffix) created by dfgtool, then generates a full coverage of each of the data flow functions by subtours. If a single subtour covers all the labels in a given function, it is this subtour which is used. If there is more than one subtour which covers all the labels then the longest subtour is (arbitrarily) selected, otherwise testgen selects more than one subtour for full coverage. This way a definition-use coverage of all context variables is assured[20], i.e. the resulting test sequences contain all the transitions that first define a variable (a D-node) and then use it, and this is true for all the variables. Definition in the DFG is characterized by an incoming arc and use by an outgoing arc. Since we generate the subtours that cover all the arcs, the definition-use coverage is thus achieved.

As an example, the disconnection function of the example protocol contains the arcs labelled by:

03 04 07 08 09 010 011 012 013 20 21 23 24

and this flow could be completely covered by seven subtours out of which we list a single one:

idle ns.ndataind [ap.transfer__cr] 31
closed map.transfer__cr [ts.tconind] 05

waittconresp ts.tdisreq [map.transfer__dr] 07
idle ap.transfer__dr nil 4
idle nil [ns.ndatareq] 23
closing map.term [ts.tdisconf] 013

TTCN test cases

Linear test sequences generated by testgen expose the interactions that occur at internal interaction points. However, nondeterminism in protocol specifications require special processing, therefore these sequences are not suitable to be represented as test cases in TTCN, the standard test suite notation. Recently, we have developed a tool that derives TTCN test steps from normalized transitions and test cases from the subtours. In converting a normalized transition into a test step, spontaneous transitions are considered to provide alternative test events. To these alternatives a *timeout* event alternative is added should the spontaneous transition contain a delay clause. Constraints for the receive and send events are generated from the PROVIDED clauses and the assignments in the actions[21].

APPLICATIONS AND PERFORMANCE

Contest-Estl has been used to generate (linear) test sequences for TP2, FTAM[22], ISDN LAPD[16] and ISDN Network layer[23]. TTCN test cases are generated for TP2 and ISDN LAPD[16]. We discuss some performance measures of non-interactive components of the tool on these applications. Thereafter we give similar measures for the TTCN test cases.

Table 1 summarizes the performance of the compiler component on TP2, FTAM and LAPD applications. The runtime is an average of 10 compilations. Table 2 summarizes the performance of the normalizer for the same three applications. The runtime is a summation of the runtimes of the first and second phases of normalization.

For the two applications where TTCN test cases were generated, Table 3 gives the number of transitions after normalization, after simplification, number of data flow functions and number of (TTCN) test cases. In Table 4 the runtime measures for simplification, multiple module tour generation (mstourgen), TTCN test step, and finally TTCN test case generation are given.

The tables show that the tool takes time proportional to the number of transitions and PDUs and ASPs in the input specification. Table 4 indicates that mstourgen is slow in cases of considerably large specifications.

CONCLUSIONS

A formal specification-based test sequence generation tool is presented. This prototype tool handles modular specifications by graphically displaying the control and data flow graphs of each module of the specification as well as generating the test sequences. The tool has recently been used to generate test sequences for several real protocols, such as TP2, FTAM and ISDN Q.921 and Q.931. Several performance measures on these applications indicate that the non-interactive parts of the tool are fast enough to effectively help the test designer in interactively designing test cases.

Extensions of the tool to cover aspects of Estelle such as system process/activity and priority clause could be achieved by improvements on two components: edittour for transition priorities, and mstourgen for parallel transition execution. Further research is needed in determing the importance of these extensions for conformance testing, which is the main application domain of the tool. It is also interesting to investigate how the tool could be extended to handle application layer protocols. Further development on the tool could be done by rewriting the normalization in a language such as C to increase its execution time performance, and porting the

Table 1. Compiler module performance

Input specification	*Size of input (lines)*	*Size of output (syntax tree, bytes)*	*Runtime (seconds)*
TP2	786	19,051	5.43
FTAM	1,750	41,904	11.93
LAPD	2,769	73,356	16.35

Table 2. Normalizer performance

Input specification	*No. of transitions before*	*No. of transitions after*	*Runtime (seconds)*
TP2	24	86	226.353
FTAM	36	103	212.932
LAPD	135	410	721.539

Table 3. Results of the application of the tool on LAPD and TP2

Specification	*No. of transitions after normalization*	*No. of transitions after simplification*	*No. of data flow functions*	*No. of test cases*
TP2	86	88	11	53
LAPD	410	490	19	1094

Table 4. Runtimes of several steps in TTCN test case generation(s)

Specification	*Simplification*	*mstourgen*	*Test step*	*Test case*
TP2	2.5	14	19.8	1.6
LAPD	17.5	9334	48.3	36.0

tool to other platforms such as personal computers would increase acceptability among the user community.

ACKNOWLEDGEMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1 *ISO/IEC JTC1/SC21*, 'Estelle: A Formal Description Technique Based on an Extended State Transition Model', IS 9074, ISO, Geneva, Switzerland (November 1988)

2 *ISO/IEC JTC1/SC21/WG1*, 'Conformance Testing Methodology and Framework', DIS 9646, ISO, Geneva, Switzerland (1989)

3 **Sarikaya, B, von Bochmann, G and Cerny, E** 'A test design methodology for protocol testing', *IEEE Trans. Softw. Eng.* (May 1987) pp 531–540

4 **Vuong, S T, Chau, A C and Chan, R I** 'Semi-automatic implementation of protocols using an Estelle-C compiler' *IEEE Trans. Softw. Eng.* (March 1988), pp 384–393

5 National Bureau of Standards, *User Guide for the NBS Prototype Compiler for Estelle*, ICST/SNA-87/3 (October 1987)

6 **Courtiat, J-P** 'Estelle and Petri nets: a PETRI NET based semantics for Estelle', in **Diaz, M, Ansart, J-P, Courtiat, J-P, Azema, P and Chari, V (eds.)**, *The Formal Description Technique Estelle – Results of the SEDOS Project*, North Holland (January 1989)

7 **Jard, C, Groz, R and Monin, J-F** 'Development of VEDA: a prototyping tool for distributed algorithms', *IEEE Trans. Softw. Eng.* (March 1988)

8 **Boyce, T T and Probert, R L** 'Phase-directed testing of Estelle specifications', in **deMeer, J, Mackert, L and Effelsberg, W (eds.)**, *Protocol Test Systems*, North Holland (1990) pp 319–326

9 **Warren, D H D** 'Logic programming and compiler writing', *Softw.-Pract. & Exper.*, Vol 10 (1980) pp 97–125

10 **Clarke, L A and Richardson, D J** 'Applications of symbolic evaluation', *J. Syst. & Software*, Vol 5 No 1 (1985)

11 **Sterling, L and Shapiro, E** *The Art of Prolog*, MIT Press, MA, USA (1986)

12 **Sidhu, D P** 'Protocol verification via executable logic specifications', *IFIP PSTV III* (June 1983) pp 237–248

13 **Ural, H and Probert, R L** 'Step-wise validation of communication protocols and services', *Comput. Networks & ISDN Syst.* Vol 11 No 3 (1986) pp 183–202

14 **Barbeau, M and Sarikaya, B** 'A computer-aided design tool for protocol testing', *INFOCOM'88* (April 1988) pp 86–95

15 **Koukoulidis, V** *Full Implementation of a Protocol Test Design Methodology*, MSc Thesis, Concordia University (March 1989)

16 **Forghani, B** *Automatic Test Suite Derivation from Estelle Specifications*, MSc Thesis, Concordia University (February 1990)

17 SUN-MICROSYSTEMS, *Programmer's Reference Manual for Sunwindows* (1985)

18 **Sarikaya, B and von Bochmann, G** 'Synchronisation and specification issues in protocol testing' *IEEE Trans. Commun.*, Vol COM-32 No 4 (1984) pp 389–395

19 **Dahbura, A, Sabnani, K, and Uyar, U** 'Formal methods for generating protocol test sequences', *IEEE Proc.* (1990)

20 **Ural, H** 'Test sequence selection based on static data flow analysis', *Comput. Commun.*, Vol 10 No 5 (1987) pp 234–242

21 **Forghani, B and Sarikaya, B** 'Automatic dynamic behavior derivation from Estelle specifications', in **deMeer, J, Mackert, L and Effelsberg, W (eds.)**, *Protocol Test Systems*, North Holland (1990)

22 **Barbeau, M, Eswara, S, Koukoulidis, V and Sarikaya, B** 'FTAM test design using an automated test tool', *Proc. INFOCOM 89*, Ottawa, Canada (April 1989)

23 **Amalou, M, and von Bochmann, G** *Conformance Test Design for ISDN D-Channel Q.931 Signalling Protocol from an Estelle Specification*, Technical Report, University of Montreal (1990)