Analysis and representation of test cases generated from LOTOS

P Tripathy*† and B Sarikaya‡

This paper presents a method to generate, analyse and represent test cases from protocol specification. The language of temporal ordering specification (LOTOS) is mapped into an extended finite state machine (EFSM). Test cases are generated from EFSM. The generated test cases are modelled as a dependence graph. Predicate slices are used to identify infeasible test cases that must be eliminated. Redundant assignments and predicates in all the feasible test cases are removed by reducing the test case dependence graph. The reduced test case dependence graph is adapted for a local single-layer (LS) architecture. The reduced test cases for the LS architecture are enhanced to represent the tester's behaviour. The dynamic behaviour of the test cases is represented in the form of control graphs by inverting the events, assigning verdicts to the events in the enhanced dependence graph.

Keywords: abstract data types, conformance testing, dependence graph, LOTOS, program slice

Increasing use of the formal description techniques LOTOS[1], Estelle[2] and SDL[3] for the specification of complex distributed systems has created considerable interest in the derivation of test cases from the specification for the purpose of testing implementations for conformance to their specifications. In the last decade, a large number of algorithms have been proposed to design test suites from formal specifications. The emphasis in most of these algorithms is placed on the fault detection capability and optimization of the generated test suites. However, the test architectures have not been taken into consideration in the design of test suites. Moreover, the generated test cases have never been analysed. In this paper, the emphasis is placed on the development of new algorithms, which are general in nature to generate, analyse and represent the test cases through a more efficient formulation of the formal model.

Earlier Milner's Chart, a particular kind of EFSM, has been used to generate test cases from the LOTOS specification[4]. In this paper, we model the generated test case by a dependence graph, which is similar to program dependence graphs[5]. The test case dependence graph is then evaluated by taking a predicate slice from it. Slicing is the abstraction of a set of statements that influence the value of a variable at a particular location. The notion of slice, originally introduced by Mark Weiser[6], is useful in program debugging, automatic parallelization and program integration. In this paper, we use the concept of slicing with respect to a predicate to detect infeasible test cases. A test case is infeasible when it contains a path from the specification that is unexecutable, in the sense that the predicate in that path can never be satisfied whatever constraints are imposed on the input event. Hence, some additional effort is needed to avoid having some test cases corresponding to infeasible paths. Identification of an infeasible (feasible) test case is a difficult task, posing many complex problems. In fact, the general problem is undecidable. Therefore, to circumvent this problem it is necessary to consider heuristics with minimum human interaction. Hence, in this paper we assume that infeasible test cases can be eliminated by evaluating predicate slices. The predicate slice of a test case with respect to a predicate consists of all statements of a test case whose execution might affect the value of that predicate. The feasible test cases modelled as a test case dependence graph can be reduced by eliminating extraneous statements that cannot affect the parameters of the event, or the control flow of the test case.

The reduced test case dependence graph represents the behaviour of a particular fragment of the protocol specification. This behaviour has to be transformed into test suites, which comprise the tester's behaviour. The tester's behaviour is the dual of the protocol entity, therefore the tester's behaviour can be obtained by behaviour inversion of the reduced test case dependence graph. Also, the predicates in the reduced test

*Bell-Northern Research, PO Box 3511, Station C, Ottawa, Ontario, Canada K1Y 4H7
†Formerly with: Acceptance Testing Group, Eicon Research Inc., 2196-32nd Avenue (Lachine), Montreal, Quebec, Canada H8T 3H7
‡Department of Computer and Information Sciences, Bilkent University, Bilkent, Ankara, 06533, Turkey
Paper received: 12 October 1993; revised paper received: 23 March 1994

case dependence graph can be eliminated in the inversion process, because its presence is insignificant. However, predicates are useful in generating input test data.

The rest of the paper is structured as follows. In the next section, the concept of EFSM is introduced. Generation and analysis of the test cases are then discussed, and the representation of test cases given. Other work related to ours is summarized and, finally, conclusions drawn.

PRELIMINARIES

This section introduces EFSM. An EFSM is similar to a finite state machine (FSM), with the extensions that a set of variables is added to the FSM, an enabling condition is associated with each transition in the machine, and a set of assignment functions are executed while firing a transition.

EFSM model

We define an extended finite state machine as: $M = <S, V_v, s_o, S_f, R, \varepsilon>$, where S is a set of states, V_v is a set of data declarations (variables), s_o is the set's initial state, S_f is a set of final states, R is the set of transitions (or rules), and ε is a set of initial value assignments to some variables V_v.

A *transition* in M is a seven-tuple: $r = <a, s, s', n, p, c, f>$, where a is an action (the event clause of r), s is the 'from' state of r, s' is the 'to' state of r, n is the transition number of r, p is the guard clause of r, c is the condition clause of r, and f is a set of assignment functions of the transition.

A transition n occurs when the EFSM is in control state s and the predicate p is true for the current assignment of the variables; then it may participate in an event that matches the action a if the condition c is satisfied. This leads to the new control state s'. The boolean condition c is introduced to capture the *selection predicate* features of LOTOS specification, whereas the condition p is to capture the guard feature of LOTOS.

In this paper, we classify the events (actions) that allow us to distinguish input (receive) and output (send) events. Question marks denote inputs, while exclamation marks denote outputs. Note that a LOTOS specification may not necessarily make this distinction, but it helps to read the specification.

LOTOS specification and its EFSM model

A LOTOS specification contains two major sections: *type definition* and *behaviour expression*. The data types are declared using Abstract Data Types[7], and behaviour expression is described using the algebraic theory of processes, based on Milner's Calculus of Communicating Systems (CCS)[8]. Behaviour description is essentially a hierarchy of nested processes that interact using gates.

We first state some assumptions on a LOTOS specification. The methodology to derive an EFSM from a LOTOS specification consists of two steps: simplification of the specification; and application of a translation rule to each operator in the specification.

Roughly speaking, for any LOTOS behaviour expression, we can construct an EFSM. Unfortunately, derivation of EFSM often diverges[9]. To avoid divergence, we have to impose some restriction on the specifications. First, we need to define some terminology. A *guard* is an external visible event, said to be an exit guard if it precedes an exit or a free guard if it precedes a free identifier X. For example, X is a free identifier in the left operand of the parallel composition in $a; (X|[S]|\mu Y.(b; Y))$, where μ is a fixed point operator used for process declaration. The event a is an exit guard in $a; exit$ but not in $a; b; exit$. Similarly, the event a is a free guard in $a; X$ but not in $a; b; X$. However, the event b is a free guard in $a; b; X$ as well as an exit guard in $a; b; exit$. A free occurrence of X in B is guarded in B if it occurs within some subexpression $a; B'$ of B. For example, X is guarded in $a; X$ but neither in X nor in $a; X[]X$. Operands of the general parallel operator are said to be synchronous if the free guard and exit guard synchronize. For example, in $X := a; b; exit|[b]|b; X$, the operands are synchronous, since they synchronize at b, which is a free guard in $b; X$ and an exit guard in $a; b; exit$.

We treat LOTOS specifications satisfying the following requirements:

1. If $\mu X.B$ is a subexpression of the process, then X is guarded in B (μ is a fixed point operator used for process declaration).
2. Operands of the general parallel operator are either closed or synchronous.
3. Operands of the pure interleaving operator are closed.

To be able to use a few simple rules to translate a LOTOS specification to an EFSM, and to avoid any conflict in the use of the same variable names in different processes, the following simplification steps are applied to a specification. First, a behaviour expression containing a full synchronization operator is transformed into a generalized parallel composition expression. Second, a sequential composition expression is simplified to a parallel composition expression, allowing us to use one rule to translate parallel and sequential expressions to EFSMs. Third, each process instantiation is replaced by its corresponding definition until the recursion point. Finally, variables in the processes are uniquely renamed.

Conceptually, a simplified LOTOS specification is rooted at its main behaviour with process names and behaviour composition operators constituting the internal nodes, the events in the specification constituting the arcs, and the *exit*, *stop* and recursive processes constituting the leaf nodes. The translation algorithm scans the simplified specification tree bottom-up. Initially, when the algorithm starts with the leaf

nodes, a simple partial EFSM with only one state and no transition is generated for each leaf node. If the leaf node is a process name, then the state is tagged with the process name to resolve the recursion at a future instant of time. As the algorithm scans the specification tree bottom-up, the partial EFSMs are updated and merged by using a translation rule for each composition operator[10]. Conceptually, each translation rule derives all possible sequential behaviour from two operand behaviours related by the operator. One needs translation rules only for the operators not eliminated during the simplification process.

We use an i_s event (spontaneous) in EFSM to distinguish the internal event in the specifications from the internal event due to the hide construct. In other words, an i_s event in EFSM is due to the internal event i in the LOTOS specifications, whereas an i event in the EFSM is due to the hide construct of the LOTOS specifications.

Example specification

A formal specification of the Association Control Service Elements (ACSE)[11] protocol in LOTOS is taken as the main example. The specification consists of two main parts. The first part describes abstract data types and structures used by ACSE, i.e. protocol data units (PDU), abstract service primitives (ASP) and their parameter types. The behaviour of ACSE is specified using a mixture of resource and state oriented styles. It has a total of some 2297 lines of LOTOS code, out of which 88% is about abstract data types. The behaviour part of the specification is given in Appendix A. An EFSM is automatically constructed from this specification[12]. The resulting EFSM has 113 states and 169 transitions.

GENERATION AND ANALYSIS OF TEST CASES

This section deals with the generation and analysis of test cases. First, test cases are generated from the EFSM, then they are analysed to detect infeasible test cases. Reductions are carried out in the feasible test cases to remove redundant assignments and predicates.

Generation of test cases

A test generated from a deterministic finite state machine consists of a sequence of test events. However, with a nondeterministic protocol model, a test case cannot be represented by a pure sequence of events, because to correctly judge the behaviour of such a protocol a test case must contain expected alternative behaviour due to nondeterminism in the protocol.

The algorithm we proposed earlier[4] to generate test cases from the EFSM takes nondeterminism into consideration. An outline of that algorithm is as follows. First, a transition tour of the EFSM is generated. The tour is divided into sequences that start from the initial state and end either in the initial state or one of the final states. The sequence is called a *partial test case*. The partial test case may contain spontaneous transition (i_s transitions). Next, check if there exists a spontaneous transition which is not present in the partial test case, but is an alternative to any one of the transitions in the partial test case. Then update the partial test case by adding a sequence of transitions such that the sequence starts with a spontaneous transition and ends either in a final state or in a state belonging to the partial test case. The procedure is repeated until no spontaneous transitions exist as alternatives to the updated partial test case. The partial test case is then a complete test case.

The application of the above algorithm to the ACSE EFSM yields 44 test cases. One complete test case, t_{28}, is shown in Listing 1. In the following test case, each tuple $< \ldots >$ represents a transition that is defined above.

```
< A?x29, 256, 121, 1, true, [IsAASCreq(x29), ε >,
< P!ACSE_apdu(ACSE_apdu_genere_0(AARQ_apdu(BIT(1),
app_context_name(get_AASCreq(x29)), called_ap_title(get_
  AASCreq(x29)),
called_ae_qualifier(get_AASCreq(x29)),
called_ap_invocation_id(get_AASCreq(x29)),
called_ae_invocation_id(get_AASCreq(x29)),
calling_ap_title(get_AASCreq(x29)),
calling_ae_qualifier(get_AASCreq(x29)),
calling_ap_invocation_id(get_AASCreq(x29)),
calling_ae_invocation_id(get_AASCreq(x29),
type_genere010(Not_Present), user_info(get_AASCreq
  (x29))))):ACSE_apdu, 121, 120, 4, true, true, ε >,
< P?x14:ACSE_apdu, 120, 109, 9, true, [IsAARE(x14)], ε >,
< i, 109, 103, 14, [eq(result(get_AARE(x14)),accepted)],
  true, ε >,
< A!primitive(AASCcnf(application_context_name(get_
  AARE(x14)),
responding_AP_title(get_AARE(x14)), responding_AE_
  qualifier(get_AARE(x14),
responding_AP_invocation_id(get_AARE(x14)),
responding_AE_invocation_id(get_AARE(x14)), user_
  information(get_AARE(x14)),
result(get_AARE(x14)), acse_service_user,
optional(result_source_diagnostic(get_AARE(x14))),
empty_presentation_parms_set)): primitive, 103, 102, 22, true,
  true, c12 ← calling >,
< A?x12 primitive, 102, 90, 30, true, [IsARLSreq(x12)], ε >,
< P!ACSE_apdu(ACSE_apdu_genere_2(RLRQ_apdu
  (reason(get_ARLSreq(x12)),
user_info(get_ARLSreq(x12))))):ACSE_apdu), 90, 89, 40,
  true, true, c10 ← c12 >,
< P?x10:ACSE_apdu, 89, 77, 53, true, [IsRLRQ(x10)], ε >,
< A!primitive(ARLSind(ARLSind(reason(get_RLRQ(x10)),
user_information (get_RLRQ(x10)))) :primitive, 77, 76, 66,
  true, true, ε >,
< i, 76, 74, 84, [eq(c10,calling)], true ε >,
< A?x7:primitive, 74, 67, 101, true, [IsAABRreq(x5)], ε >,
< P!ABRT_apdu(acse_service_user,type_genere023(Not_
  Present)):ABRT_apdu, 67, 256, 116, true, true, ε >.
```

Listing 1 t_{28} test case

The test case listed consists of 12 transitions. There are no spontaneous transitions (i_s transitions). The assignment function of the transition is denoted by $y \leftarrow E$, which means that the value of expression E is assigned to the variable y. The empty assignment clause is denoted by ε, whereas empty guard and condition clauses are denoted by a 'true' value.

Test case dependence graph

In this section, we define the test case dependence graph (TCDG) in terms of the control flow graph of a test case. The control flow graph $CG = (V, E, en)$ of a test case is a directed graph having a unique entry node en. V is a set of nodes corresponding to assignments (s-node), actions (a-node) and predicates (p-node). Graphically, a-, p- and s-nodes are represented by a circle, triangle and a rectangle, respectively. E is a set of control edges which represents a possible transfer of control from one node to another. The control edge from node v_i to node v_j is denoted by $v_i \rightarrow_c v_j$.

The TCDG of a test case is the control graph of the test case with the addition of data dependence edges. A data dependence edge from node v_i to node v_j implies that the computation performed at node v_i directly depends upon the value computed at node v_j. More precisely, it means that the computation performed at node v_i uses a variable, *var*, that is defined at node v_j, and there is an execution path from v_j to v_i along with the variable; *var* is not (re-)defined. The data dependence edge from node v_i to node v_j is denoted by $v_i \rightarrow_d v_j$.

Formally, a test case dependence graph for a test case t is a digraph $G_t = (V_t, E_t, en)$ with $V_t = V_a \cup V_s \cup V_p$, $E_t = E_d \cup E_c$, and a unique entry node $en \in V_t$, where $V_a = \{v | v \text{ is an a-node}\}$; $V_s = \{v | v \text{ is an s-node}\}$; $V_p = \{v | v \text{ is a p-node}\}$; $E_d = \{(u, v) | u \rightarrow_d v\}$; $E_c = \{(u, v) | u \rightarrow_c v\}$.

We provide an algorithm to construct a control flow graph $CG = (V, E, en)$ of a generated test case t. Before that we need some definitions. We assume that the assignment clause f of any transition r is in the form of a tuple $f = <f_1, f_2 \ldots, f_n>$, where each f_i is of the form $y \leftarrow E(x_1, x_2, \ldots, x_n)$, where E is a value expression containing variables $x_1, x_2, \ldots, x_n$. For any transition $r \in t$, we define the following functions:

1. From[r] returns the from clause of r.
2. Action[r] returns the action clause of r.
3. To[r] returns the to clause of r.
4. Transition[r] returns the transition number of r.
5. Guard[r] returns the guard clause of r.
6. Condition[r] returns the condition clause of r.
7. Assignment[r] returns the assignment clause of r.
8. f_i(assignment[r]) returns the ith item of the assignment clause $f = <f_1, f_2, \ldots, f_n>$.
9. Initial[t] returns the initial state of the test case t, which is nothing but the initial state of the EFSM M.

Algorithm: Test case control graph construction.
Input: Test case t in the form of EFSM transitions R_t.
Output: Test case control graph $CG = (V, E, en)$.
We can assume that appropriate data structures are available to create different types of nodes and arcs. Also available in the structure is a place for a label for each node. We use two functions, last_node and first_node, for each transition to keep track of the last and first nodes of the transition:

S1. For all $r \in R_t$ do mark 'new';
S2. Let $R_{initial[t]} := \{r | r \in R_t, \text{from}[r] = \text{initial}[t]\}$.
S3. For all $r \in R_{initial[t]}$ in R_t do mark 'old'.
S4. Construct the initial node *en* of the test case control graph.
S5. DRAW_CONTROL($R_{initial[t]}, R_t, en$).

Procedure DRAW_CONTROL(R, R_t, ev);

S1. For each $r \in R$ do
- (i) Set last_node[transition[r]] := ev and first_node[transition[r]] := $\emptyset$;
- (ii) If guard[r] < > 'true' then do
  - (a) Create a p-node 'u' with label guard[r];
  - (b) Construct a control edge from last_node [transition[r]] to the p-node 'u';
  - (c) Set last_node[transition[r]] := u and first_node[transition[r]] := u.
- (iii) If action[r] < > i then do
  - (a) Create an a-node 'u' with label action[r];
  - (b) Construct a control edge from last_node [transition[r]] to the a-node 'u';
  - (c) Set last_node[transition[r]] := u;
  - (d) If first_node[transition[r] = $\emptyset$ then first_node[transition[r]] := u.
- (iv) If condition[r] < > 'true' then do
  - (a) Create a p-node 'u' with label action[r];
  - (b) Construct a control edge from last_node [transition[r]] to the p-node 'u';
  - (c) Set last_node[transition[r]] := u;
  - (d) If first_node[transition[r]] = $\emptyset$ then first_node[transition[r]] := u.
- (v) If assignment[r] < > ε then for each assignment statement $f_i \in <f_1, f_2, \ldots, f_n>$, where $1 \leqslant i \leqslant n$ do
  - (a) Create an s-node 'u' with label f_i(assignment[r]);
  - (b) Construct a control edge from last_node [transition[r]] to the s-node 'u';
  - (c) Set last_node[transition[r]] := u;
  - (d) If first_node[transition[r]] = $\emptyset$ then first_node[transition[r]] := u.

S2. For each $r \in R$ do
- (i) Let $T_{to[r]} := \{s | s \in R_t \text{ and from}[s] = \text{to}[r]\}$.
- (ii) If every $s \in T_{to[r]}$ in R_t is marked 'old' then for each $s \in T_{to[r]}$ construct a control edge from last_node[transition[r]] to first_node [transition[s]] else mark each $t \in T_{to[r]}$ in R_t as 'old' and call DRAW_CONTROL ($T_{to[r]}$, R_t, last_node[transition[r]]).

Each variable occurrence in a graph CG is classified as being a definition or use[13]. We use the following convention to identify definition and use of each variable in CG:

1. An action $?x:u$ in an a-node contains definition of variable x.
2. An action $!E(x_1, x_2, \ldots, x_n)$ in an a-node contains uses of $x_1, \ldots, x_2$, where E is a value expression containing variables $x_1, x_2, \ldots, x_n$.
3. An assignment statement $y \leftarrow E(x_1, x_2, \ldots, x_n)$ in an s-node contains uses of $x_1, \ldots, x_2$ followed by a definition of y, where E is a value expression containing variables $x_1, x_2, \ldots, x_n$.
4. A predicate $p(x_1, x_2, \ldots, x_n)$ in a p-node contains uses of $x_1, x_2, \ldots, x_n$.

Based on the above classification, we can create a set of data dependence edges in the test case control flow graph CG.

Algorithm: Test case dependence graph construction.
Input: Test case control graph $CG = (V, E, en)$.
Output: Test case dependence graph $G_t = (V, E \cup E_d, en)$.

For each $v \in V_t$ do

S1. Evaluate $X_v := \{x | x \text{ is a definition in } v\}$.
S2. For each $x \in X_v$ do the following. If there exists a node $u \in V$, where x is used and u can be reachable from v through the control edges along which x is not (re-)defined, then create a data dependence edge from node u to node v.

The event i in EFSM is due to the hide construct of the LOTOS specifications. There will be no alternative to this event in the test case dependence graph. It is not the real internal event of the LOTOS specifications, therefore the event i may be suppressed in the dependence graph. However, a spontaneous transition (i_s transition) is represented by an a-node, which plays an important role in the representation of a test case. As an example, consider the test case t_{28}. *Figure 1* shows its test case dependence graph. For convenience, the nodes are numbered as follows. First the transition number is placed, followed by a period and the tuple number. For a-nodes the tuple number is 1, for p-nodes 5 or 6 (depending on whether it is in the guard or condition clause of the transition). Similarly, for s-nodes the tuple number is 7.

Predicate slices

The predicate slice of a test case with respect to a predicate, *pred* at a p-node, consists of all nodes whose execution could possibly affect the boolean value of *pred* at the p-node. The predicate slice of a p-node can be constructed easily by traversing the data dependence edges of the test case dependence graph beginning at p-node. The nodes visited during traversal constitute the desired slice. We will provide an algorithm to get all the predicate slices from the test case dependence graph. Before that we need some definitions.

Given a node $v \in V_t$, we define the set $D_f[v]$, $D_t[v]$, $C_f[v]$, $C_t[v]$, $N_f[v]$ and $N_t[v]$ as follows. $D_t[v] = \{(u, v) | (u, v) \in E_d\}$; $D_f[v] = \{(u, w) | (v, w) \in E_d\}$; $C_t[v] = \{(u, v) | (u, v) \in E_c\}$; $C_f[v] = \{(u, w) | (v, w) \in E_c\}$; $N_t[v] = \{u | (u, v) \in E_c\}$; $N_f[v] = \{w | (v, w) \in E_c\}$.

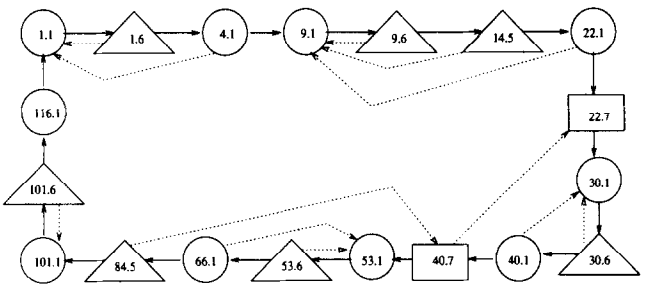

Figure 1 Test case dependency graph for t_{28}

For a p-node p of a test case dependence graph G_t, the predicate slice of G_t with respect to p, denoted by $G_{t/p}$, is a graph containing all nodes on which p has a data dependence (i.e. all nodes that can reach from p via a data dependence edge): $V(G_{t/p}) = \{w | w \in V_t \text{ and } p \xrightarrow{*}_d w\}$. We extend the definition to the set of all p-nodes $V_p = \cup_i p_i$ as follows. $V(G_t/V_p) = V(G_t/\cup_i p_i) = \cup_i V(G_t/p_i)$. The edges in the graph G_t/V_p are essentially those in the subgraph G_t induced by $V(G_t/V_p)$, with the restriction that only data dependence edges are included. We define $E(G_t/V_p) = \{(v, w) | (v \xrightarrow{*}_d w) \in E_d \text{ and } v, w \in V(G_t/V_p)\}$.

Algorithm: Predicate slices.
Input: Test case dependence graph $G_t = (V_t, E_t)$.
Output: Two sets $V' = V(G_t/v_p)$ and $E' = E(G_t/v_p)$, which represent predicate slices for each p-node $v_p \in V_t$.

The recursive procedure slice(v) adds edge (v, w) to E' if node w is first reached during the search by a data dependence edge from v. For each p-node, all the nodes are marked 'new' and procedure SLICE is invoked.

```
1. Let V_p be the set p-nodes in V_t;
2. V' := V_p;
3. E' := φ;
4. for each v_p in V_p do
     begin
5.     for all v in V_t do mark v 'new';
6.     SLICE(v_p);
     end.
```

procedure SLICE(v);

```
1. add {v} to V';
2. mark v 'old';
3. for each edge (v, w) on D_f[v] do
     begin
4.     if w is marked 'new' then
       begin
5.       add (v, w) to E';
6.       SLICE(w);
       end;
     end.
```

The time complexity of the algorithm given above in the worst case is $O(|V_p|(|V_t| + |E_t|))$. Lines 1 and 5 of

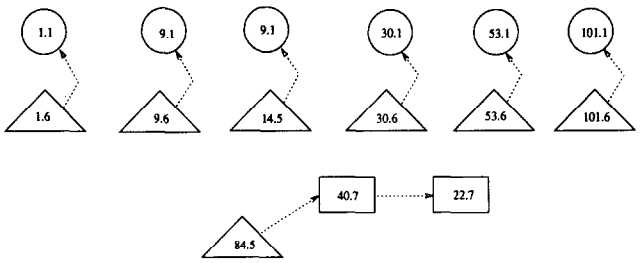

Figure 2 Predicate slices of the test case dependency graph t_{28}

Predicate Slices take time $O(|V_t|)$. Lines 5–6 are called exactly $|V_p|$ times. The time spent in SLICE is exclusive of recursive calls to itself, proportional to $|D_t[v]|$. Since $\sum_{v \in V_t} |D_t[v]| = O(|E_t|)$, the total cost of executing lines 3–5 of SLICE is $O(|E_t|)$. The procedure SLICE is called exactly once for each vertex $v \in V_t$, since v is marked 'old' the first time SLICE is called. Thus, the total time spent in Predicate Slices is $O(|V_p|(|V_t| + |E_t|))$.

Figure 2 shows the graph that results from taking predicate slices of the test case dependence graph from *Figure 1*.

Classification of Predicate Slices

We classify the predicate slices into three classes, and discuss their role in protocol testing:

(a) *Determinable Predicate Slices.* In communication protocols, there are two kinds of determinable predicates:

- The static predicates.
- The dynamic predicates.

The static predicates are those which concern variables whose values do not change. These types of predicates are generated in the EFSM due to the value matching type of interaction in resolving parallel composition. The dynamic predicates are those whose values are changed by an assignment clause of the rule following a specific event. A predicate slice is said to be a determinable predicate slice if it contains either a static or dynamic predicate. For example, the p-node 84.5 of *Figure 2* is a determinable predicate, hence the predicate slice is a determinable predicate slice.

(b) *Undeterminable Predicate Slices.* The undeterminable predicates are those that concern variables whose values are generated nondeterministically. This happens in the case of value generation, e.g. generalized choice constructs and hiding of input events. These predicates can be neither completely controlled by the IUT nor by the tester. A predicate slice is said to be an undeterminable predicate slice if it contains an undeterminable predicate.

(c) *Settable Predicate Slice.* The settable predicates are those that depend upon the parameter value of the input primitive. A predicate slice is said to be a settable predicate slice if it contains a settable predicate. These predicates can be set to true by choosing the proper values for the parameters of the input primitives. For example, the p-nodes 1.6, 9.6, 14.5, 30.6, 53.6 and 101.6 of *Figure 2* are settable predicates, hence the predicate slices are settable predicate slices.

Infeasible paths in test cases

Predicate plays an important role in the evaluation of test cases generated from the protocol specification. Some of the test cases generated may not be feasible, in other words, the predicates can never be satisfied on a path due to the existence of an assignment on that path that will cause the predicate to be set to false. Infeasible paths in a test case can be detected by evaluating the determinable predicate slices.

To correct infeasibilities we have to exchange the transition containing the unsatisfiable predicate with a new transition, which means that the path following the old transition must be eliminated and a new path has to be selected following the new transition. The result of this is two-fold: the new path is the same as one of the test cases already generated, where no new test case is added; otherwise, the new path becomes one of the test cases. In both cases, the infeasible test cases are eliminated.

In the determinable predicate slice of *Figure 2*, the determinable predicate node 84.5 is dependent on the assignment node 40.7, which is in turn dependent on the assignment node 22.7, and evaluates to true. The reason is that at s-node 22.7, c12 is assigned to 'calling', and then at s-node 40.7 it is assigned to c10, i.e. now c10 has the value of 'calling'. The predicate at p-node 84.5 is evaluated to be true because c10 is assigned the value 'calling'. Out of 44 test cases generated from the ACSE protocol, seven of them were found to be infeasible.

Reduction of test cases

Once the test case is modelled by a dependence graph, it can be reduced by eliminating extraneous statements that cannot affect the parameters of the event or the control flow of the test case. The s-nodes in the TCDG with no incoming data dependence edges can be eliminated, since these nodes neither affect the parameters of the events nor the control flow of the test case. Since infeasibilities are already removed before this step, we can reduce further the TCDG by eliminating all p-nodes from which a-nodes are not reachable through the data dependence edges. For example, in the determinable predicate slice of *Figure 2b*, no a-nodes can be reachable starting from the p-nodes 84.5. Clearly, these p-nodes have no *influence* on the input/output domain of the test case. In other words, we keep all the settable predicates in the test case dependence graph that depend upon the parameter values of the input primitives. Intuitively, redundant assignments and predicates are those that may be executed, but its elimination would not change the function of the test case computed over its domain. The resulting test case

dependence graph obtained after elimination of redundant assignments and predicates is called the reduced test case dependence graph (RTCDG).

Algorithm: Reduced test case dependence graph.
Input: Test case dependence graph $G = (V, E)$.
Output: Reduced test case dependence graph $G' = (V', E')$.

The recursive procedure SEARCH eliminates an s-node that has no data dependence edge incident on it. The main algorithm eliminates all the p-nodes from which a-nodes are not reachable through the data dependence edges.

1. $V' := V$;
2. $E' := E$;
3. Let $I_p := \{v \in V_p | v \not\rightarrow_d^* w$, where $w \in V_a\}$;
4. for each $v \in I_p$ do
   begin
5. $V' := V' - v$;
6. Let $E_{new} := \{(u, w) | w \in N_f[v]$ and $u \in N_t[v]\}$;
7. $E' := \{E' - \{D_f[v] \cup C_f[v] \cup C_t[v]\}\} \cup E_{new}$;
   end;
8. SEARCH(V', E').

procedure SEARCH(V', E');

1. Let $VD = \{v | D_t[v] = \phi$ and $v \in V'_s \subset V'\}$;
2. If $VD <> \phi$ then
   begin
3. Let v be any element in VD;
4. $V' := V' - v$;
5. Let $E_{new} := \{(u, w) | w \in N_f[v]$ and $u \in N_t[v]\}$;
6. $E' := \{E' - \{D_f[v] \cup C_f[v] \cup C_t[v]\}\} \cup E_{new}$;
7. SEARCH(V', E')
   end;

The time complexity of the algorithm given above in the worst case is $O(|V||V_s|)$. The cost of executing line 3 of RTCDG can be $O(|V||V_s|)$. The loop on the line 5–7 of RTCDG is executed $|V_p|$ times. The total cost of executing SEARCH, exclusive of recursive calls to itself, is $O(|V|)$. The procedure SEARCH is invoked $|V_s|$ times, each time one vertex $v \in V_s$ is deleted. Thus, the total time spent in SEARCH is $O(|V||V_s|)$. Hence, the time complexity of the RTCDG algorithm is $O(|V||V_s|)$.

The reduced test case dependence graph of test case t_{28} is shown in *Figure 3*.

TEST SUITE REPRESENTATION

According to the LS test architecture of the ISO, any implementation under test (IUT) can be tested by a lower tester (LT) and upper tester (UT) located at the bottom and top interfaces, respectively (see *Figure 4*)[14]. Test cases generated from the specification define the behaviour of the IUT. The behaviour of the LT and UT considered together comprises the tester's behaviour. The tester's behaviour is the dual of the IUT's behaviour, therefore the tester's behaviour can be obtained by behaviour inversion. Behaviour inversion is based on viewing each test case as an extended finite

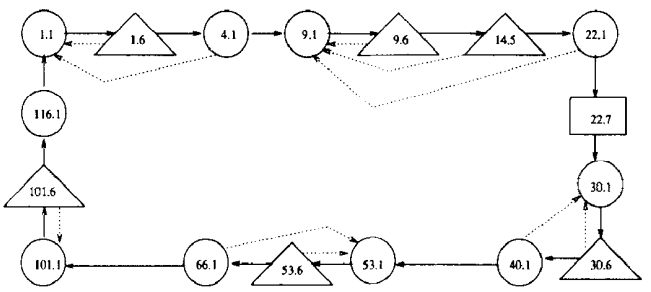

Figure 3 Reduced test case dependency graph t_{28}

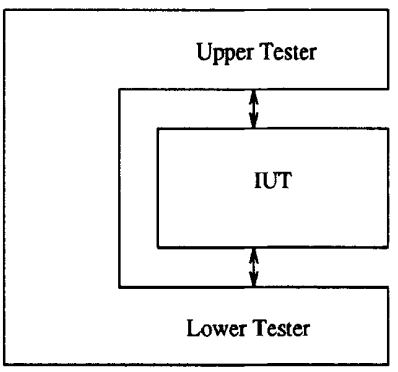

Figure 4 LS test architecture

state machine. Complete behaviour generation of the test case EFSMs is a complex process consisting of several steps. We have already discussed the steps of specification transformation, test case generation and test case reduction. In this section, we represent dynamic behaviour of the test case.

Control flow behaviour representation

Events and assignments in a test case comprise the dynamic behaviour of the test case. The settable predicates are useful in generating input test data, but its presence in the dynamic behaviour of the test case will be redundant, hence, it can be eliminated.

The flow of control is sequential except when there is spontaneous transition. Graphical representation of the control flow can be directly obtained from RTCDG by simply dropping the settable predicate p-nodes and the associated s-nodes with no incoming data dependence edge. The settable predicates are useful in generating input test data, but its presence in the dynamic behaviour of the test case will be redundant, hence it can be eliminated. Except for i_s a-nodes, all other nodes are inverted, which means that events are inverted. In other words, input events are converted to output events, and *vice versa*.

Algorithm: Control flow behaviour representation (CFBR).
Input: Reduced test case dependence graph RTDCG $= (V, E)$.
Output: Control flow behaviour representation CFBR $= (V', E')$.

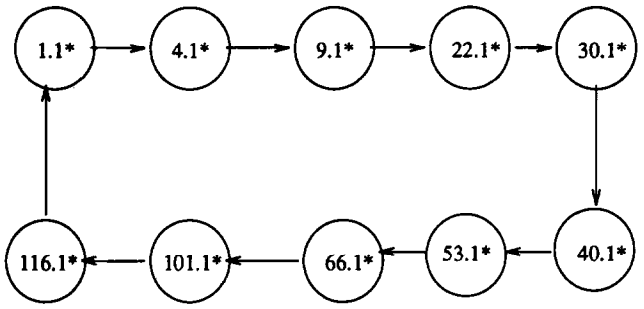

Figure 5 Control flow behaviour representation of the RTCDG of t_{28}

The main algorithm eliminates all the p-nodes. The actions are inverted. The procedure SEARCH is the same as defined in the reduced test case dependence graph algorithm.

1. $V' := V$;
2. $E' := E$;
3. for each $v \in V_p \subset V$ do
   begin
4. $V' = V' - v$;
5. Let $E_{new} := \{(u, w) | w \in N_f[v]$ and $u \in N_t[v]\}$;
6. $E' := \{E' - \{D_f[v] \cup C_f[v] \cup C_t[v]\}\} \cup E_{new}$;
   end;
7. SEARCH(V', E');
8. for each $v \in V'_a \subset V'$ of the form $gd_1, \ldots, d_n$ do
   begin
9. for $i = 1$ to n do
   begin
10. if d_i is in the form $!t_i$ then change it to $?u_i : s_i$
11. else if d_i is in the form of $?u_i : s_i$ then change it to $!t_i$
   end;
   end.

The graphical control flow behaviour representation of the test case t_{28} is given in *Figure 5*. The '*' in the a-nodes represents the inversion.

Behaviour enhancements

Here, control flow behaviour representation (CFBR) of the selected test cases is analysed, and several enhancements are carried out. In the following, we outline an algorithm to enhance CFBR of the selected valid behaviour test case:

Algorithm: Behaviour enhancement.
Input: Control flow behaviour representation (CFBR).
Output: Enhanced CFBR.

A new type of receive a-node (?OTHERWISE) is created as an alternative to all receive a-nodes to specify a tester's behaviour against invalid IUT behaviour. Verdicts are assigned to the receive and OTHERWISE a-nodes.

S1. for each receive a-node do
add an alternative path which contains an arc and a receive a-node of type OTHERWISE. No other arcs are added to this path, i.e. OTHERWISE a-nodes can only be at the final states.

S2. For each OTHERWISE a-node O1 do verdict(O1) := fail;
For each receive a-node R1 do
If R1 is the last event in the path and the path leads to the initial state then verdict(R1) := pass;

S3. For each i_s a-node A1 do
for each receive a-node R2 following A1 do
if R2 is the final node or predecessor of the final node then verdict(R2) := inconclusive.

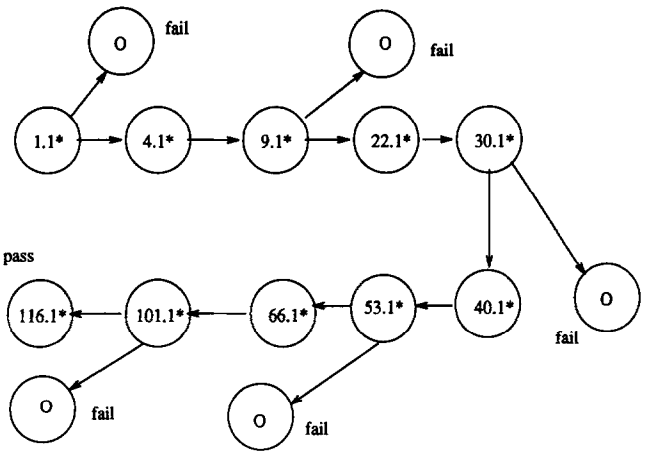

Figure 6 Enhanced CFBR of the selected test case t_{28}

Applying the behaviour enhancement algorithm, a pass verdict is associated with the a-node 116.1*, and OTHERWISE a-nodes are added to become alternatives to the a-nodes 4.1*, 22.1*, 40.1*, 66.1* and 116.1*. Step 2 assigns fail verdicts to all these OTHERWISE a-nodes. The test starts at node 1.1* and ends successfully at a-node 116.1*, or unsuccessfully at any OTHERWISE a-node. The enhanced CFBR of the test case t_{28} is shown in *Figure 6*.

RELATED RESEARCH IN TEST SUITE DESIGN

Designing test cases (suites) from formal specifications is an active area in protocol testing. Research in this area is inspired by the rich results obtained previously in hardware/software testing. Nevertheless, it is evolving towards its own set of techniques, tools and disciplines, possibly due to the distinct characteristics of protocols and their architectures. In the last decade, a large number of algorithms have been proposed to design test suites from formal specification. We will discuss the existing protocol test case design based on Estelle, LOTOS and SDL.

Estelle-based test design

Since deterministic FSMs model only the control component of the protocol/services, there is a need to extend the FSM-based techniques to cover the data

component, i.e. interaction primitive processing and data transfer mechanisms. Recently, two methods have been proposed to design test cases from EFSM. The first method[17,18] is based on data flow analysis techniques[19], and focuses on tracing the flow of data through the associations between assignments of values to variables and references of these variables in either assigning values to other variables or determining the outcome of conditional branching. The second method[20] applies the principles of functional testing[21]. In the functional testing method, two different graphs are obtained from the normalized specification: a control graph for major state change, and a data flow graph to show the flow of data from input service primitive/protocol data unit parameters to the context variables, and from the context variables to output service primitive/protocol data unit parameters. The control graph is an FSM, from which test cases are generated. The data flow graph is partitioned into blocks, where each block corresponds to functions of the protocol. The test suite design with this methodology is based on obtaining the control sequence for a test case and enumerating the parameters of the interaction primitives. Each block is tested with one or more test cases until all the arcs in the partitioned data flow graph are covered. The resulting tests are used as behaviour tests for establishing the dynamic conformance testing.

A semi-automatic test generation method, which considers the context and predicates of specifications written in Estelle is described by Favreau and Linn[22]. In this method, tests are generated in two steps. First, a state machine is created from an abstract machine and an initial context, where an abstract machine is a kind of FSM extended with predicates and variables. Second, the transition-tour technique is applied to the FSM generated in the first step. Most of the semi-automatic methods have an advantage in that they include human intuition to express the test's purposes, which is an important aspect of conformance testing[23].

Recent results in this area are reported elsewhere[24–26]. In Miller and Paul[24], weak mutation testing is adopted in contrast to functional testing, whereas in Wang and Liu[25], a program verification technique called *axiomatic semantics* is applied to the conformance testing area. In Lee and Lee[26], the issue of testing data flow is not addressed. They consider the problem of generating a control flow graph for a protocol entity specified as a collection of communicating modules.

The emphasis in most of the algorithms discussed above is placed on the generation of test cases from the Estelle specifications. However, the test cases generated have never been analysed in the design of test suites. In our approach, emphasis is given to analysis of the test cases generated. This paper supplements recent investigations on the design of test suites by presenting a formal, general model for representation of the generated test cases. The analysis presented in this paper, by modelling a test case as a test case dependence graph, combines traditional control flow analysis and data flow analysis, hence it can be implemented efficiently.

LOTOS-based test design

There has been much research on test suite generation from LOTOS specifications. The first related work can be found in Brinksma[27] and Wezeman[28], where the derivation of the conformance testers T(S) for any specification S has been investigated. In Brinksma[27], a failure model is used to identify processes that are testing equivalently, whereas in Wezeman[28], a syntactical approach is explored, based on the work reported in Steenbergen[29]. The method is named the CO-OP method after its main components, the set called COmpulsory and OPtional behaviours.

Recent results in this area have been reported[30–32]. In Tretmans[32], a conformance relation *conf* is defined to establish whether an implementation under test (IUT) conforms to its specification. The *conf* relation can be explained as follows. If $B1$ and $B2$ are processes, then $B1$ *conf* $B2$ if $B1$ contains no unexpected deadlocks with respect to traces in $B2$. However, it does not allow $B1$ to possess traces not specified in $B2$. For example $B1 = (a; exit[]i; b; exit[]c; exit)$ conforms to $B2 = (a; exit[]i; b; exit)$ even though $B1$ has a trace $(c; exit)$ that is not included in $B2$. But it is not the case that $B3 = (a; exit[]i; exit)$ *conf* $B2$. This relation is taken as a formal basis for extending the CO-OP method to a restricted form of full LOTOS. Process definition and recursion of the LOTOS are not considered. Only choice, the action prefix and a limited action denotation are considered. The method only uses the COmpulsory set, unlike in the CO-OP method, where COmpulsory and OPtional sets are used. In Wezeman *et al.*[30,31], the CO-OP method is extended to a restricted subset of full LOTOS that handles events involving the exchange of data. The CO-OP method is used in deriving conformance testers in the ISO Conformance Testing Methodology and Framework standard[14].

In related work[33], where manually an FSM model is derived using a LOTOS interpreter[35], classical methods are then applied to generate test cases. The authors have not yet considered the data flow aspect with the interpretation approach. Recently, a group of researchers at Neher laboratories[34] proposed an interactive methodology to generate test cases from LOTOS specification. A symbolic evaluation is used to derive test cases. However, the methodology is not algorithmic in nature. It is similar to the method proposed by Gueraichi and Logrippo[35].

SDL-based test design

Generation of a test suite from SDL has been studied by Hogrefe[36]. In this model, SDL processes are transformed into an intermediate form called the asynchronous communication tree (ACT). The ACT is a tree

where the root is the initial system state, the nodes are all other system states, and the arcs represent the valid transition. The tree is constructed by considering in turn every possible distinct event that may occur at each node. The arc is labelled with the event, and the node is identified by the resulting process states and queue contents. The depth of the tree is restricted by comparing the current state with the states generated previously; if an identical state is found, that branch is terminated. Finally, this ACT is traversed to generate test cases. Nondeterminism has not been taken into consideration in the generation of test cases. The test generation algorithm proposed in this paper, which deals with nondeterminism, can be used to generate a test suite from an ACT.

Several test generation tools[36, 37] based on the SDL specification have been developed. The tools assume a single module and use the Abstract Syntax Notation One (ASN.1)[38] description of data units as the data input format.

CONCLUSIONS

The objective of this paper is twofold: first, to introduce an intermediate model, and to use this as an interim step for the generation of tests from protocol specifications; and second, to represent the dynamic behaviour of the test case generated. To achieve the first goal, we have demonstrated that LOTOS can be transformed into EFSM, and have shown that the test cases can be generated easily from the EFSM.

To achieve our second goal, the test cases generated are modelled as a test case dependence graph and then evaluated by taking predicate slices from it. The predicate slices are classified into three classes: determinable, undeterminable and settable. These three classes of predicate slices are analysed to evaluate their role in protocol testing. Redundant assignments along with determinable and undeterminable predicates in all the feasible test cases are removed by reducing the test cases. Finally, the control flow behaviour is represented by inverting the direction of the actions, assigning verdicts to the actions, and eliminating all the settable predicates from the reduced test case dependence graph. We have demonstrated the applicability of this method by applying it to a LOTOS specification of the ACSE protocol. The methodology is schematically shown in *Figure 7*.

The nature of research is such that the solution to one problem often gives rise to many new questions or problems. In the case of the research presented in this paper, the following questions surface naturally. Further investigation could be the generation of input test data. The settable predicate slices may be useful for that purpose. The input test data must be generated from these settable predicates such that the predicates evaluated are true.

Representation of the test cases in the form of an abstract test suite for different architectures used in practice, such as distributed, coordinated and remote test architectures, is another line of future research. Finally, it would be interesting to automate the methodology discussed here.

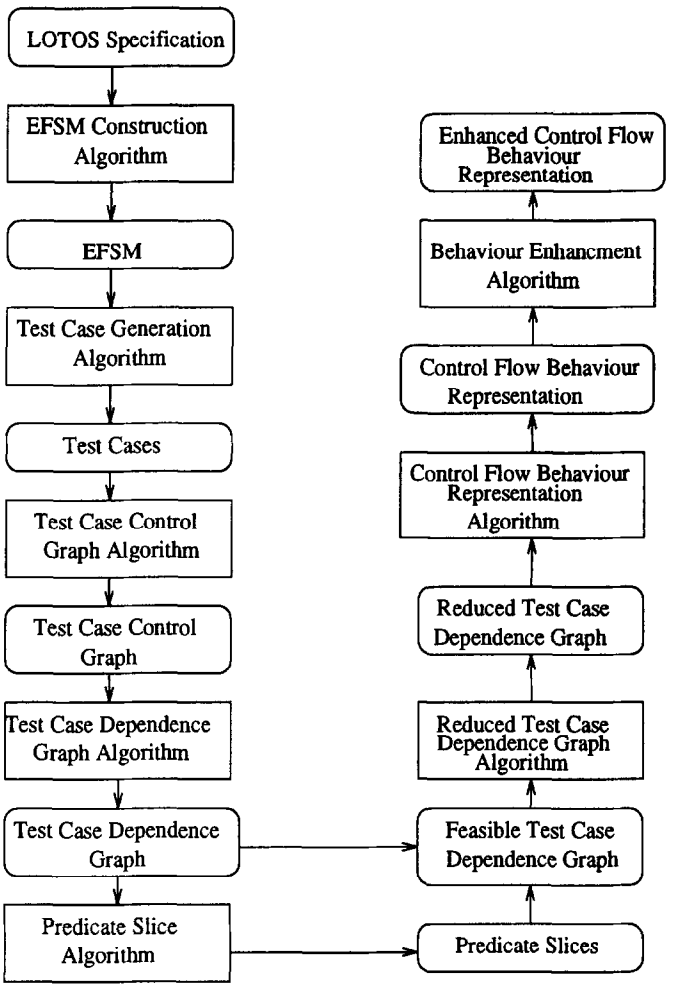

Figure 7 Test suite derivation methodology

REFERENCES

1 Bolognesi, T and Brinksma, E 'Introduction to the ISO specification language LOTOS', *Comput. Networks ISDN Syst.*, Vol 14 (1987) pp 25–59

2 Budkowski, S and Dembinski, P 'An introduction to Estelle: a specification language for distributed systems', *Comput. Networks ISDN Syst.*, Vol. 14 (1987) pp 3–23

3 Belina, F and Hogrefe, D 'The CCITT-specification and description language SDL', *Comput. Networks ISDN Syst.*, Vol 16 (1989) pp 311–341

4 Tripathy, P and Sarikaya, B 'Test case generation from LOTOS specification', *IEEE Trans. Comput.*, Vol 40 No 4 (1991) pp 543–552

5 Ferrante, J, Ottenstein, K J and Warren, J D 'The program dependence graph and its uses in optimization', *ACM Trans. Program. Lang. Syst.*, Vol 9 No 3 (July 1987) pp 319–349

6 Weiser, M 'Program slicing', *IEEE Trans. Softw. Eng.*, Vol 10 No 4 (1984) pp 352–357

7 Ehrig, H and Mahr, B *Fundamentals of Algebraic Specification*, Springer-Verlag, Berlin (1985)

8 Milner, R *A Calculus of Communicating Systems: Lecture Notes in Computer Science*, Vol 92, Springer-Verlag, Berlin (1980)

9 Karjoth, G 'Implementing process algebra specifications by state machines', *IFIP PSTV VIII*, Atlantic City, NJ (June 1988)

10 Tripathy, P *A Unified Model for Protocol Test Suite Design*, PhD Thesis, Concordia University (November 1992)

11 *Protocol Specification for the Association Control Service Elements*, ISO DIS8650, ISO, Geneva, Switzerland (January 1988)
12 Tripathy, P and Sarikaya, B *LOTEST: A LOTOS Test Case Generation Tool*, Tech. Rep. (1992)
13 Frankl, P G and Weyuker, E J 'An application family of data flow testing criteria', *IEEE Trans. Softw. Eng.*, Vol 14 (1988) pp 1483–1489
14 *Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework*, ISO/IEC 9646, ISO, Geneva, Switzerland (1991)
15 Sidhu, D P and Leuing, T-K 'Formal methods for protocol testing: detailed study', *IEEE Trans. Softw. Eng.*, Vol 15 No 4 (April 1989) pp 413–426
16 Dahbura, A T, Sabnani, K K and Uyar, M U 'Formal methods for generating protocol conformance test sequences', *Proc. IEEE*, Vol 78 (1990) pp 1317–1326
17 Ural, H 'Test sequence selection based on static data flow analysis', *Comput. Commun.*, Vol 10 No 5 (October 1987)
18 Ural, H and Yang, B 'A test sequence selection method for protocol testing', *IEEE Trans. Commun.*, Vol 39 No 4 (1991) pp 514–523
19 Rapps, S and Weyuker, E J 'Selecting software test data using data flow information' *IEEE Trans. Softw. Eng.*, Vol 11 (1985) pp 367–375
20 Sarikaya, B, von Bochmann, G and Cerny, E 'A test design methodology for protocol testing', *IEEE Trans. Softw. Eng.*, Vol 13 (May 1987)
21 Howden, W E *Functional Program Testing and Analysis*, McGraw Hill, New York (1987)
22 Favreau, J P and Linn, R J 'Automatic generation of test scenario skeletons from protocol specification writen in Estelle', *IFIP PSTV VI* (1986)
23 Linn, R J 'Conformance evaluation methodology and protocol testing', *IEEE Trans. Selected Areas in Commun.*, Vol 7 No 7 (September 1989) pp 1141–1158
24 Miller, R E and Paul, S 'Generating conformance test sequences for combined control and data flow of communication protocols', *IFIP PSTV XII* (1992)
25 Wang, C-J and Liu, M T 'A test suite generation method for extended finite state machines using axiomatic semantics approach', *IFIP PSTV XII* (1992)
26 Lee, D Y and Lee, J Y 'A well-defined Estelle specification for automatic test generation', *IEEE Trans. Comput.*, Vol 40 No 4 (1991) pp 526–542
27 Brinksma, E 'A theory for the derivation of tests', *IFIP PSTV VIII*, North-Holland, Amsterdam (1988) pp 63–74
28 Wezeman, C 'The CO-OP method for compositional derivation of conformance testers', *IFIP PSTV IX* (1989) pp 145–158
29 Steenbergen, C *Conformance Testing of OSI Systems*, MSc Thesis, University of Twente (1986)
30 Wezeman, C, Batley, S and Lynch, J 'Formal methods to assist conformance testing: a case study', *Proc. 3rd Int. Conf. Formal Description Techniques (FORTE'90)*, Madrid, Spain (1990)
31 Wezeman, C 'Deriving tests from LOTOS specifications', *Proc. 3rd Lotosphere Workshop and Seminar*, Pisa, Italy (September 1992)
32 Tretmans, J *A Formal Approch to Conformance Testing*, PhD Thesis, University of Twente (December 1992)
33 Gueraichi, D and Logrippo, L. 'Derivation of test cases for LAP-B from LOTOS specification', *Proc. 2nd Int. Conf. Formal Description Techniques (FORTE'89)*, Vancouver, Canada (1989)
34 van de Burgt, S P, Kroon, J, Kwast, E and Wilts, H J 'The RNL conformance Kit', *Proc. IFIP T6 Second Int. Workshop Protocol Test Systems*, Berlin, Germany (October 1989)
35 Logrippo, L, Obaid, A, Braind, J P and Fehri, M C 'An interpreter for LOTOS, a specification language for distributed systems', *Softw.–Practice Exper.*, Vol 18 No 4 (April 1988) pp 365–385
36 Hogrefe, D 'Automatic generation of test case from SDL specification', *SDL Newsletter*, No 12 (June 1988)
37 Katsuyama, K, Sato, F, Nakakawaji, T and Mizuno, T 'Strategic testing environment with formal description techniques', *IEEE Trans. Comput.*, Vol 40 (April 1991) pp 514–523
38 *Profile of Abstract Syntax Notation-one*, ISO 8824, ISO, Geneva, Switzerland (1987)

APPENDIX A

```
behaviour
ACSE[A,P]
where

process ACSE[A,P] : noexit :=
unassociated[A,P] [>protocol_error[A,P]
endproc (*ACSE*)

process unassociated[A,P] : noexit :=
  A ? x : primitive [IsAASCreq(x)];
        P !ACSE_apdu(ACSE_apdu_genere_0(AARQ_apdu(
        Bit(1),app_context_name(get_AASCreq(x)),
        called_ap_title(get_AASCreq(x)),
        called_ae_qualifier(get_AASCreq(x)),
        called_ap_invocation_id(get_AASCreq(x)),
        called_ae_invocation_id(get_AASCreq(x)),
        calling_ap_title(get_AASCreq(x)),
        calling_ae_qualifier(get_AASCreq(x)),
        calling_ap_invocation_id(get_AASCreq(x)),
        calling_ae_invocation_id(get_AASCreq(x)),
        type_genere010(Not_Present),user_info(get_AASCreq(x)))));
        awaitAARE[A,P]
  []
  P ? x : ACSE_apdu [IsAARQ(x)];
  (
  [Not( common_prot_version(get_AARQ(x)) )] ->
     P !ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(
       Bit(1),
       application_context_name(get_AARQ(x)),
       rejected_permanent,
       Associate_source_diagnostic(Associate_source_diagnostic_genere_1
                                   (no_common_acse_version)),
       type_genere013(Not_Present),
```

```
      type_genere014(Not_Present),
      type_genere015(Not_Present),
      type_genere016(Not_Present),
      type_genere017(Not_Present),
      type_genere018(Not_Present))));
       unassociated[A,P]
    []
 [common_prot_version(get_AARQ(x))] ->
 A !primitive(AASCind(
   optional(normal),
   application_context_name(get_AARQ(x)),
   calling_AP_title(get_AARQ(x)),
      calling_AE_qualifier(get_AARQ(x)),
      calling_AP_invocation_id(get_AARQ(x)),
      calling_AE_invocation_id(get_AARQ(x)),
      called_AP_title(get_AARQ(x)),
      called_AE_qualifier(get_AARQ(x),
      called_AP_invocation_id(get_AARQ(x)),
      called_AE_invocation_id(get_AARQ(x)),
      user_information(get_AARQ(x)),
      empty_presentation_parms_set));
      awaitAASCrsp[A,P]
   )
where
      process awaitAARE[A,P] : noexit :=
       P ?x : ACSE_apdu [IsAARE(x))];
           (
            [result(get_AARE(x)) eq accepted] ->
                   A !primitive(AASCcnf(
                   application_context_name(get_AARE(x)),
                   responding_AP_title(get_AARE(x)),
                   responding_AE_qualifier(get_AARE(x)),
                   responding_AP_invocation_id(get_AARE(x)),
                   responding_AE_invocation_id(get_AARE(x)),
                   user_information(get_AARE(x),
                   result(get_AARE(x)),
                   acse_service_user,
                   optional(result_source_diagnostic(get_AARE(x))),
                   empty_presentation_parms_set));
                   associated[A,P](calling)
           []
            [result(get_AARE(x)) eq rejected_permanent] ->
                   A !primitive(AASCcnf(
                   application_context_name(get_AARE(x)),
                   responding_AP_title(get_AARE(x)),
                   responding_AE_qualifier(get_AARE(x)),
                   responding_AP_invocation_id(get_AARE(x)),
                   responding_AE_invocation_id(get_AARE(x)),
                   user_information(get_AARE(x),
                   result(get_AARE(x)),
                   acse_service_user,
                   optional(result_source_diagnostic(get_AARE(x))),
                   empty_presentation_parms_set));
                   unassociated[A,P]
           )
            [> abort[A,P]
 endproc (* awaitAARE *)
 process awaitAASCrsp[A,P] : noexit :=
          A ? x : primitive [IsAASCrsp(x)];
          (
           [result(get_AASCrsp(x)) eq accepted] ->
            P !ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(
                   Bit(1),
                   app_context_name(get_AASCrsp(x)),
                   result(get_AASCrsp(x)),
                   Associate_source_diagnostic(Associate_source.
                    diagnostic_genere_0(no_reason_given)),
                   responding_ap_title(get_AASCrsp(x)),
                   responding_ae_qualifier(get_AASCrsp(x)),
                   responding_ap_invocation_id(get_AASCrsp(x)),
                   responding_ae_invocation_id(get_AASCrsp(x)),
                   type_genere017(Not_Present),
```

```
                    user_info(get_AASCrsp(x)))));
              associated[A,P](called)
            []
             [not( result(get_AASCrsp(x)) eq acccepted )] ->
              P !ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(
                    Bit(1),
                    app_context_name(get_AASCrsp(x)),
                    result(get_AASCrsp(x)),
                    Associate_source_diagnostic(Associate_source.
                     diagnostic_genere_0(no_reason_given)),
                    responding_ap_title(get_APPSCrsp(x)),
                    responding_ae_qualifier(get_AASCrsp(x)),
                    responding_ap_invocation_id(get_AASCrsp(x)),
                    responding_ae_invocation_id(get_AASCrsp(x)),
                    type_genere017(Not_Present),
                    user_info(get_AASCrsp(x)))));
              unassociated[A,P]
                )
            [> abort[A,P]
  endproc (* awaitAASCrsp *)
endproc (* unassociated *)
  process associated[A,P](c : calltype) : noexit :=
            P ? x : ACSE_apdu [IsRLRQ(x)];
            A !primitive(ARLSind(reason(get_RLRQ(x)),
               user_information(get_RLRQ(x))));
             awaitARLSrsp[A,P](c)
       []
           A ? x : primitive [IsARLSreq(x)];
            P !ACSE_apdu(ACSE_apdu_genere_2(RLRQ_apdu(reason(get_ARLSreq(x)),
                                                user_info(get_ARLSreq(x)))));
              awaitRLRE[A,P](c)
          [> abort[A,P]
  where
            process awaitRLRE[A,P](c : calltype) : noexit :=
              P ? x : ACSE_apdu [IsRLRE(x)];
(* On suppose qu'on a recu le RLRE dans un PCONcnf+ *)
               A !primitive( ARLScnf(reason(get_RLRE(x)),
                    user_information(get_RLRE(x)),
                          accepted));
                 unassociated[A,P]
               []
                P ? x : ACSE_apdu [IsRLRQ(x)];
                 A !primitive(ARLSind(reason(get_RLRQ(x)),
                     user_information(get_RLRQ(x))));
                   (
                   [c eq called] ->
                       collision_associated_responder[A,P]
                  []
                     [c eq calling] ->
                       collision_association_initiator[A,P]
                   )
                   [> abort[A,P]
               where
                 process collision_association_initiator[A,P] : noexit :=
                         A ? x : primitive [IsARLSrsp(x)];
                           (
                         [result(get_ARLSrsp(x)) eq accepted] ->
                           P !ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu(reason
                               (get_ARLSrsp(x)), user_info(get_ARLSrsp(x)))))
                           awaitRLRE[A,P](calling)
                            )
                              [> abort[A,P]
                    endproc (* collision_association_initiator *)
                    process collision_association_responder[A,P) : noexit :=
                            P ? x : ACSE_apdu [IsRLRE(x)]
(* On suppose que le RLRE provient d'un PCONcnf+ *)
                           A !primitive(ARLScnf(reason(get_RLRE(x)),
                               user_information(get_RLRE(x)),
                                     accepted));
                            awaitARLSrsp[A,P](called)
                           [> abort[A,P]
                    endproc (* collision_association_responder *)
```

```
            endproc {* awaitRLRE *)
            process awaitARLSrsp[A,P](c : calltype : noexit :=
                    A ? x : primitive [IsARLSrsp(x)];
                    (
                      [result(get_ARLSrsp(x)) eq accepted] ->
                          P !ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu(reason
                            (get_ARLSrsp(x)), user_info(get_ARLSrsp(x)))));
                          unassociated[A,P]
                    []
                      [result(get_ARLSrsp(x)) eq rejected] ->
                          P !ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu(reason
                            (get_ARLSrsp(x)), user_info(get_ARLSrsp(x)))));
                          associated[A,P](c)
                    )
                      [> abort[A,P]
            endproc {* awaitARLSrsp *)
         endproc (* associated *)
       process (abort[A,P] : noexit :=
            A ? x : primitive [IsAABRreq(x)];
            P !ABRT_apdu(acse_service_provider,type_genere023(Not_Present));
            unassociated[A,P]
         []
            P ? x : ACSE_apdu [IsABRT(x)];
            A !AABRind(acse_service_provider,type_genere020(Not_Present));
            unassociated[A,P]
      endproc (* abort *)
      process protocol_error[A,P] : noexit :=
            A !AABRind(acse_service_provider,type_genere020(Not_Present));
            P !ABRT_apdu(acse_service_provider,type_genere023(Not_Present));
            ACSE[A,P]
      endproc (* protocol_error *)
endspec
```