Automatic Test Case Generation for Estelle*

Chang-Jia Wang and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277, USA

Abstract

In this paper, an automatic test case generation method for Estelle is proposed. A formal model is introduced to describe the dynamic properties of Estelle specifications so as to verify the difference between the behavior of the specification and the behavior of its fault models. Based on the difference, an algorithm is presented to produce test cases that can detect such implementation faults. The algorithm can generate test cases not only for single module specifications, but also for systems containing multiple modules that run concurrently. In addition, heuristics are suggested to improve the performance of the test case generation process.

1 Introduction

Protocol conformance testing is a major issue in the design of reliable communication networks. It ensures that a *protocol implementation* is consistent with its *specification* in various hardware and software environments. One major issue of conformance testing is *test case generation*. Most of the test case generation methods proposed are based on the *Finite State Machine (FSM)* model [1, 2, 3, 4]. However, the FSM model can only specify problems within the domain of regular languages. To solve more general problems, other models such as the *Extended Finite State Machine (EFSM)* [5], *Estelle* [6] or *LOTOS* [7] are used [8, 9, 10, 11, 12, 13, 14]. These Models usually contain memories. Therefore, in addition to testing control flow, the data aspect of the models needs to be considered as well.

A test case is meaningless if one does not know its purpose. A *test purposes* is a set of statements showing what type of error a test case tries to detect. The error type is called a *fault model* [15], which must be given before any meaningful test case can be generated. The test methods mentioned above are used to generate test cases for a fixed fault-model. When one needs the confidence that certain critical faults will not occur, these methods cannot guarantee the detection of the cirtical faults if they are not included in the fixed fault-models.

It is infeasible to generate a test case for every possible fault. Thus, it is desirable to provide the protocol designer with a freedom of choosing whatever faults he/she believes are important. Based on our previous work [13, 14], a test case generation method for EFSM that produces test cases for given fault models [16] has been developed. The method employs a program verification technique, called axiomatic semantics [17, 18, 19], to symbolically verify a protocol specification. The difference between the behavior of the specification and the behavior of the fault model is analyzed and then test cases are generated based on the difference.

Since Estelle has been adopted as an international standard specification language for communication protocols, it is interesting to extend the proposed method in [16] from EFSM to Estelle. In this paper, an automatic test case generation method for Estelle is proposed. After axioms for Estelle statements are defined, the same algorithm in [16] can be used to generate test cases for given fault models. In addition, the same method is extended to generate test cases for multiple modules running concurrently. Furthermore, a heuristic search method is also suggested to improve the performance of the search algorithm.

The rest of this paper is organized as follows: In Section 2, the behavior model proposed in [16] is briefly introduced. Then, axioms for Estelle statements are defined in Section 3, and the definition of a test case and how to generate it are presented in Section 4. Finally, conclusions are given in Section 5.

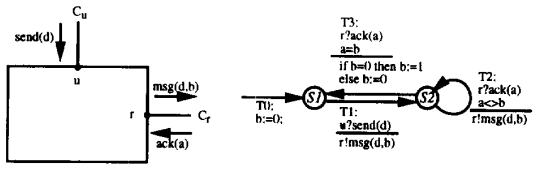
2 Background

2.1 Estelle and EFSM

Estelle [6] is a description language for *Extended Finite State Machines (EFSM)*. An EFSM is a *Finite State Machine (FSM)* with memory (called *variables* hereafter). Similar to an FSM, an EFSM contains a set of *states* and *transitions* pointing from one state (called *head state*) to another (called *tail state*). There are usually a *condition* and an *action* associated with each transition. The action of a transition can be executed only when the EFSM is at

*Research reported herein was supported by U.S. Army Research Office, under contracts No. DAAL03-91-G-0093 and No. DAAL03-92-G-0184. The views, opinions, and/or findings contained in this paper are those of the authors and should not be construed as an official Department of the Army position, policy or decision.

0-8186-3670-X/93 $03.00 © 1993 IEEE



(a)

```
initialize
to S1
name T0
    begin
    b:=0
    end;

from S2
to S2
when r.ack(a)
provided ab
name T2
    begin
    output r.msg(d,b)
    end;
```

(b)

```
from S1
to S2
when u.send(d)
name T1
    begin
    output r.msg(d,b)
    end;

from S2
to S1
when r.ack(a)
provided a=b
name T3
    begin
    if b=0 then b:=1
    else b:=0
    end;
```

(c)

Figure 1: The sender of an ABP protocol

the head state of the transition and the condition is satisfied. After the action is executed, the EFSM moves to the tail state of the transition and becomes ready for executing next transition. In Estelle, the head and tail states are defined in **from** and **to** clauses, respectively. The condition of a transition includes **when** and **provided** clauses. The former inputs a message from an *interaction point (IP)* and the latter imposes a restriction under which the action can be executed. The action is a Pascal statement sequence quoted between reserved words **begin** and **end**. The statements can be executed only when both the input in **when** clause is available and the condition in **provided** clause is satisfied. Readers who are not familiar with the Estelle syntax are referred to [6].

In addition to the clauses described above, there are declaration statements and other clauses that are not important for generating test cases. Therefore, in the rest of this paper, we focus only on the portion of specification that defines the transitions, and assume that all modules, channels, IPs, and variables are properly defined.

A sample Estelle specification is shown in Figure 1, which is the sending side of an *Alternating Bit Protocol (ABP)*. The structure of the protocol is shown in Figure 1a, the state diagram is shown in Figure 1b, and corresponding Estelle transitions are defined in Figure 1c.

2.2 Behavior of an EFSM

Estelle is able to clearly describe the structure of an EFSM. However, knowing the structure alone is inadequate for generating test cases. In order to find test cases for an EFSM, one needs to know the *dynamic property* of the EFSM as well. Specifically, the knowledge of how the variables in the EFSM change their values is needed. Therefore, the *behavior model* proposed in [16] is used in this paper to describe such changes.

When a transition of an EFSM is executed, the values of the variables change accordingly. The values are preserved by their *versions*, which provide names to store different values of the same variable during different periods of time. The versions are denoted by an operator ν (pronounced "new") followed by the names of the variables. For example, νx denotes a newer version of variable x. Similarly, $\nu\nu x$ (or simply $\nu^2 x$) denotes a newer version of νx, and $\nu^0 x$ (or simply x) denotes the original version of variable x.

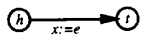
The relationship between versions and their values is called a *scenario*. Formally, a scenario is a function that maps a set of versions into a set of values. It is denoted as a sequence of versions and their corresponding values quoted by a pair of angle brackets ($\langle\ \rangle$). For example, a scenario, $\langle x = 0, y = 0, \nu x = 1, \nu y = 0, ...\rangle$, shows that x and y are originally zeros, and then change to 1 and 0 later, respectively.

An *assertion* is a set of scenarios. It is denoted by a boolean expression, called *predicate*, quoted by a pair of braces ($\{\ \}$). The assertion contains those scenarios that satisfy the predicate. For example, assertion $\{\nu y \geq 0\}$ contains both scenarios $\langle \nu x = 1, \nu y = 0, ...\rangle$ and $\langle \nu x = 1, \nu y = 1, ...\rangle$. The set of all possible scenarios that can be developed after a state, say a, is denoted by $\beta[a]$. For instance, $\beta[a] = \{\nu y \geq 0\}$ means that if the EFSM is at state a, every possible scenario developed thereafter has the property that the first version of y is greater than or equal to zero.

Let M be an EFSM. The behavior of M is denoted as $\beta[\![M]\!]$, which is defined as $\beta[\![M]\!] = \bigcup_{\forall s} \beta[s]$, where s is a state of M. In addition, let A_1 and A_2 be two assertions, and p_1 and p_2 be two predicates such that $A_1 = \{p_1\}$ and $A_2 = \{p_2\}$. It can be shown that $A_1 \cup A_2 = \{p_1 \vee p_2\}$, $A_1 \cap A_2 = \{p_1 \wedge p_2\}$, $\overline{A_1} = \{\neg p_1\}$, and $A_1 - A_2 = \{p_1 \wedge \neg p_2\}$.

2.3 Axioms

A rule that describes how a statement changes the course of the scenarios' development is called an *axiom*. Figure 2 is a simple example for the axiom of an assignment statement. The transition starts from state h, points to state t, and contains only one action: "$x := e$," which assigns the result of expression e to variable x. The axiom means that the scenarios that can be developed after state h is the same as the scenarios developed after state t with one more restriction that the new version of variable x is equal to the result of expression e (denoted by assertion $\{\nu x = e\}$). Since variable x has been upgraded, any development of variable x after state t must start from the upgraded version. Therefore, the second term of the axiom is denoted by $\beta[t]||^{\{\nu x\}}$, meaning that every



$$\beta[h] = \{\nu x = e\} \cap \beta[t]|^{\{\nu x\}}$$

Figure 2: An example of an axiom

occurrence of x in $\beta[t]$ is replaced by νx. In general, for any assertion A, $A|^{\{\nu^i x\}}$ denotes replacing each occurrence of $\nu^j x$ in A by $\nu^{i+j} x$. For example, if $A = \{x = 0 \wedge \nu x = 1\}$, then $A|^{\{\nu x\}} = \{\nu x = 0 \wedge \nu^2 x = 1\}$.

Let A_T be the assertion that indicates the restriction imposed by T, and let V_T be the set of new versions created by T. If the set of scenarios developed after state h through transition T is denoted as $\beta[h]_T$, then $\beta[h]_T = A_T \cap \beta[t_T]|^{V_T}$, meaning that the scenarios developed after h through T are those developed after t_T (with upgraded versions) that satisfy A_T. The *parallel composition* of the behavior at state h ($\beta[h]$) means that $\beta[h]$ is the combination of all the transitions fan out from h. Thus, $\beta[h]$ can be derived as follows:

$$\begin{aligned}\beta[h] = {} & \beta[h]_{T_1} \cup \beta[h]_{T_2} \cup \cdots \cup \beta[h]_{T_n} \\ = {} & (A_{T_1} \cap \beta[t_{T_1}]|^{V_{T_1}}) \cup (A_{T_2} \cap \beta[t_{T_2}]|^{V_{T_2}}) \cup \cdots \\ & \cup (A_{T_n} \cap \beta[t_{T_n}]|^{V_{T_n}})\end{aligned}$$

The *serial composition* of $\beta[h]$ means that $\beta[h]$ is the resulting scenario set derived from a sequence of transitions after state h. Therefore,

$$\begin{aligned}\beta[h] = {} & A_{T_1} \cap \beta[t_{T_1}]|^{V_{T_1}} \\ = {} & A_{T_1} \cap (A_{T_2} \cap \beta[t_{T_2}]|^{V_{T_2}})|^{V_{T_1}} \\ = {} & A_{T_1} \cap A_{T_2}|^{V_{T_1}} \cap \beta[t_{T_2}]|^{V_{T_1} \uplus V_{T_2}} \\ & \vdots \\ = {} & A_{T_1} \cap A_{T_2}|^{V_{T_1}} \cap A_{T_3}|^{V_{T_1} \uplus V_{T_2}} \cap \\ & \cdots \cap \beta[t_{T_n}]|^{V_{T_1} \uplus V_{T_2} \uplus \cdots \uplus V_{T_n}}\end{aligned}$$

where the operation "$\uplus$" is defined as follows:

Definition 1 *Let V_1 and V_2 be two sets of versions. Then*

$$\begin{aligned}V_1 \uplus V_2 = {} & \{\nu^{i+j} w \mid \exists \nu^i w \in V_1 \wedge \exists \nu^j w \in V_2\} \cup \\ & \{\nu^i w \mid \exists \nu^i w \in V_1 \wedge \forall \nu^j w \notin V_2\} \cup \\ & \{\nu^j w \mid \forall \nu^i w \notin V_1 \wedge \exists \nu^j w \in V_2\}\end{aligned}$$

$$\begin{aligned}V^n = {} & \underbrace{V \uplus V \uplus \ldots \uplus V}_{n} \\ = {} & \{\nu^{nk} w \mid \nu^k w \in V\}\end{aligned}$$

3 Axioms for Estelle

Estelle specifications can be written in *normal form specifications*, in which every transition is specified in the following format:

from h
to t
when $p.m(x_1, x_2, ..., x_n)$
provided $\mathcal{B}$
begin S **end**;

where h and t, respectively, are the head and tail states of the transition, p is an interaction point, m is a message received from p with arguments x_1 through x_n, and $\mathcal{B}$ is a boolean expression indicating the condition under which a sequence of Pascal statements, S, can be executed. A transition can be divided into two simpler transitions: the first transition contains **when** and **provided** clauses, and the second executes S. An auxiliary state u can be placed between the two transitions. Then, the axiom for an Estelle transition can be defined as follows:

Axiom 1

$$\begin{aligned}\beta[h] = {} & \{\nu c_p = m(\nu x_1, \nu x_2, ..., \nu x_n)\} \cap \\ & \{(\nu x_1 = \nu\kappa) \wedge (\nu x_2 = \nu^2\kappa) \wedge \cdots \wedge (\nu x_n = \nu^n\kappa)\} \cap \\ & \{\nu\tau = \nu c_p\} \cap \\ & \{\mathcal{B}\} \cap \\ & \beta[u]|^{\{\nu c_p, \nu x_1, ..., \nu x_n, \nu^n\kappa, \nu\tau\}}\end{aligned}$$

There are five terms in the axiom. The first term shows that the current data received from c_p (denoted νc_p) is message m with arguments νx_1 through νx_n. The values of νx_1 to νx_n are shown in the second term, in which an auxiliary variable κ is used to indicate different input values. The third term is used to arrange the external events in chronicle order, where τ is an auxiliary variable used to record current I/O events. Therefore, "$\nu\tau = \nu c_p$" indicates that the current external event is at channel c_p. The fourth term shows that the condition in $\mathcal{B}$ must be satisfied. The last term means that the scenario set developed after the transition contains those scenarios developed after state u with variable versions updated.

The remaining problem is to define the axioms for S in order to obtain $\beta[u]$ in terms of $\beta[t]$. In the following axioms, the notations A_s and V_s represent, respectively, the assertion and the set of new versions developed after executing a statement s or a sequence of statement s.

Axiom 2 (Empty Statements)
If S is empty, $\beta[u] = \beta[t]$.

Axiom 3 (Sequential Statements)
If S is "s; S'," where s is a statement and S' is a sequence of statements,

$$\beta[u] = A_s \cap A_{S'}|^{V_s} \cap \beta[t]|^{(V_s \uplus V_{S'})}.$$

Axiom 4 (Assignment Statements)
If S is "$x := e$," where x is a variable and e is an expression,

$$\beta[u] = \{\nu x = e\} \cap \beta[t]|^{\{\nu x\}}$$

Axiom 5 (Output Statements)
*If S is "***output** $p'.m'(e_1, e_2, ..., e_n)$*," where p' is an interaction point, m' is an output message, and expressions e_1 to e_n are the arguments of m', then*

$$\beta[u] = \{\nu c_{p'} = m'(e_1, e_2, ..., e_n)\} \cap \{\nu\tau = \nu c_{p'}\} \cap \beta[t]|^{\{\nu c_{p'}, \nu\tau\}}$$

where $c_{p'}$ is the corresponding channel of p'.

In Axiom 5, the first term indicates that the next event at channel $c_{p'}$ is $m'(e_1, e_2, ..., e_n)$. The second term appends the event to the sequence of external events recorded by τ.

Axiom 6 (Selection Statements)
*If S is "***if** $\mathcal{B}$ **then** S_1 **else** S_2*," where $\mathcal{B}$ is a boolean expression and S_1 and S_2 are two sequences of statements,*

$$\beta[u] = (\{\mathcal{B}\} \cap A_{S_1} \cap \beta[t]|^{V_{S_1}}) \cup (\{\neg\mathcal{B}\} \cap A_{S_2} \cap \beta[t]|^{V_{S_2}})$$

A selection statement can be viewed as a parallel composition of two transitions. One can be executed only when condition $\mathcal{B}$ is satisfied, and the other only when it is not.

Axiom 7 (Function Calls)
Let F be a function defined as follows:

Function $F(x_1, x_2, ..., x_n)$;
begin S_F; $F := e$ **end;**

where $x_1, x_2, ..., x_3$ are parameters of F, S_F is a sequence of statements, and e is an expression. If S is "$y := F(a_1, a_2, ..., a_n)$," where $a_1, a_2, ..., a_n$ are arguments, then

$$\begin{aligned}\beta[u] \;=\; & \{(\nu x_1 = a_1 \wedge \nu x_2 = a_2 \wedge ... \wedge \nu x_n = a_n)\} \cap \\ & A_{S_F}|^{V'} \cap \\ & \{\nu y = e\}|^{V_{S_F} \uplus V'} \cap \\ & \beta[t]|^{V_{S_F} \uplus V' \uplus \{\nu y\}}\end{aligned}$$

where $V' = \{\nu x_1, \nu x_2, ..., \nu x_n\}$.

The first term of the axiom indicates that the arguments (the a's) are assigned to the parameters (the x's). The assertion imposed by the body of the function (A_{S_F}) restricts the development of the scenarios after the function call. Therefore, it is included in the second term. The third term shows that after the function call, variable y receives the value from expression e (thus $\{\nu y = e\}$). Finally, the new versions generated by the above are contained in $V_{S_F} \uplus V' \uplus \{\nu y\}$, so the variables in $\beta[t]$ need to be upgraded accordingly. The axiom for procedure calls can be defined in a similar manner.

Axiom 8 (Procedure Calls)
Let P be a procedure defined as follows:

Procedure $P(x_1, x_2, ..., x_n,$ **var** $y_1,$ **var** $y_2, ...,$ **var** $y_m)$;
begin S_P **end;**

where $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$ are variables and S_P is a sequence of statements. If S is "$P(a_1, a_2, ..., a_n, b_1, b_2, ..., b_m)$," where $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ are arguments, then

$$\begin{aligned}\beta[u] \;=\; & \{(\nu x_1 = a_1 \wedge \nu x_2 = a_2 \wedge ... \wedge \nu x_n = a_n) \wedge \\ & (\nu y_1 = b_1 \wedge \nu y_2 = b_2 \wedge ... \wedge \nu y_m = b_m)\} \cap \\ & A_{S_P}|^{V'} \cap \\ & \{\nu b_1 = y_1 \wedge \nu b_2 = y_2 \wedge ... \wedge \nu b_n = y_n\}|^{V_{S_P} \uplus V'} \cap \\ & \beta[t]|^{V_{S_P} \uplus V' \uplus V''}\end{aligned}$$

where $V' = \{\nu x_1, \nu x_2, ..., \nu x_n, \nu y_1, \nu y_2, ..., \nu y_m\}$ and $V'' = \{\nu b_1, \nu b_2, ..., \nu b_m\}$.

It is assumed that every variable has its own name. Although this is not true in a real specification, name conflicts can always be resolved by attaching the corresponding function or procedure names to the variable names.

Axiom 9 (While Statement)
*If S is "***while** $\mathcal{B}$ **do** S'*," where $\mathcal{B}$ is a boolean expression and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-1} [\{\mathcal{B}\} \cap A_{S'}]|^{V_{S'}^k} \cap [\{\neg\mathcal{B}\} \cap \beta[t]]|^{V_{S'}^n}$$

Let $\beta[u]_i$ denote the behavior of state u after the i-th iteration, and assume that S' will be executed n times. Then,

$$\begin{aligned}\beta[u] \;&=\; (\{\mathcal{B}\} \cap A_{S'}) \cap \beta[u]_1|^{V_{S'}} \\ &=\; (\{\mathcal{B}\} \cap A_{S'}) \cap (\{\mathcal{B}\} \cap A_{S'})|^{V_{S'}} \cap \beta[u]_2|^{V_{S'}^2} \\ &\;\;\vdots \\ &=\; \textstyle\bigcap_{k=0}^{n-1}(\{\mathcal{B}\} \cap A_{S'})|^{V_{S'}^k} \cap \beta[u]_n|^{V_{S'}^n} \\ &=\; \textstyle\bigcap_{k=0}^{n-1}(\{\mathcal{B}\} \cap A_{S'})|^{V_{S'}^k} \cap (\{\neg\mathcal{B}\} \cap \beta[t]|^{V_{S'}^n})\end{aligned}$$

It is possible to construct a data structure to represent "$\bigcap_{k=0}^{n-1}(\{\mathcal{B}\} \cap A_{S'})|^{V_{S'}^k}$" such that n remains undefined; the actual value of n will be determined later when the specification is analyzed. Similarly, **repeat** and **for** statements can be defined as follows.

Axiom 10 (Repeat Statements)
*If S is "***repeat** S' **until** $\mathcal{B}$*," where $\mathcal{B}$ is a boolean expression and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=1}^{n} [A_{S'}|^{V_{S'}^{k-1}} \cap \{\neg\mathcal{B}\}|^{V_{S'}^k}] \cap [\{\mathcal{B}\} \cap \beta[t]]|^{V_{S'}^n}$$

Axiom 11 (For-To Statements)
*If S is "***for** $i := m$ **to** n **do** S'*," where i is an integer, m and n are integer expressions, and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-m} [\{\nu i = k + m\} \cap A_{S'}]|^{(\nu i \uplus V_{S'})^k} \cap \beta[t]|^{V_{S'}^{n-m+1}}$$

$$
\begin{aligned}
\beta[S_1] \;=\;& \{(\nu c_u = send(\nu d)) \wedge (\nu d = \nu\kappa) \wedge (\nu\tau = \nu c_u)\}\cap \\
& \{(\nu c_r = msg(\nu d, b)) \wedge (\nu^2\tau = \nu c_r)\}\cap \\
& \beta[S_2]|^{\{\nu c_u, \nu c_r, \nu d, \nu\kappa, \nu^2\tau\}} \\
\beta[S_2] \;=\;& (\{(\nu c_r = ack(\nu a)) \wedge (\nu a = \nu\kappa) \wedge (\nu\tau = \nu c_r)\}\cup \\
& \{(\nu a \neq b) \wedge (\nu^2 c_r = msg(d, b)) \wedge (\nu^2\tau = \nu^2 c_r)\}\cap \\
& \beta[S_2]|^{\{\nu a, \nu^2 c_r, \nu\kappa, \nu^2\tau\}})\cup \\
& (\{(\nu c_r = ack(\nu a)) \wedge (\nu a = \nu\kappa) \wedge (\nu\tau = \nu c_r)\}\cap \\
& \{(\nu a = b) \wedge ((b = 0 \wedge \nu b = 1) \vee (b = 1 \wedge \nu b = 0))\}\cap \\
& \beta[S_1]|^{\{\nu a, \nu b, \nu c_r, \nu\kappa, \nu\tau\}})
\end{aligned}
$$

Figure 3: The behavior function of the ABP in Figure 1

Axiom 12 (For-Downto Statements)
*If S is "***for** $i := n$ **downto** m **do** S'," *where i is an integer, m and n are integer expressions, and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-m} [\{\nu i = n - k\} \cap A_{S'}]|^{(\nu i \uplus V_{S'})^k} \cap \beta[t]|^{V_{S'}^{n-m+1}}$$

The behavior of the EFSM in Figure 1 can be derived as the equations shown in Figure 3, where $\beta[S_1]$ is derived from Axioms 1 and 5, and $\beta[S_2]$ is derived from Axioms 1, 4, 5, and 6. For simplicity, the initial transition is ignored. Substituting $\beta[S_2]$ into the first equation in Figure 3, $\beta[S_1]$ can be expanded to describe more detailed information about the behavior of the specification at state S_1. Unless every executable path ends up at a final state, the recursive functions never stop. A method to extract a finite external event sequence as a test case will be presented in the next section.

4 Test Case Generation

4.1 Functional Equivalence

Two scenarios are *functionally equivalent* if they generate the same external events. For example, scenarios

$\langle \nu a = 0, \nu d = \nu\kappa, \nu b = 1, \nu c_r = msg(\nu d, \nu b), \nu\tau = \nu c_r \rangle$, and
$\langle \nu a = 1, \nu d = \nu\kappa, \nu b = 1, \nu c_r = msg(\nu d, \nu b), \nu\tau = \nu c_r \rangle$

are functionally equivalent since both output $msg(\nu\kappa, 1)$ to channel c_r as their first external events (because $\nu\tau = \nu c_r = msg(\nu d, \nu b)$, where $\nu d = \nu\kappa$ and $\nu b = 1$). The value of νa is unimportant in the example since it does not affect the outcome. Formally, two scenarios h and g are functionally equivalent if $h(\nu^i\tau) = g(\nu^i\tau)$ for any nonnegative integer i, where $h(\nu^i\tau)$ denotes the value of $\nu^i\tau$ in scenario h.

A *functionally equivalent set (FES)* of an assertion A is a set of all functionally equivalent scenarios of the scenarios in assertion A. The FES of A, denoted $\mathbf{FES}(A)$, is defined as follows:

Definition 2
$\mathbf{FES}(A) = \{g \mid \forall i \geq 0, \exists h \in A, [h(\nu^i\tau) = g(\nu^i\tau)]\}$

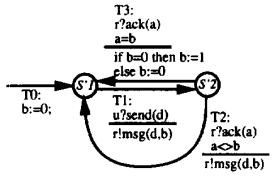


Figure 4: Two mutants for the EFSM in Figure 1

For example, if assertion

$A = \{\nu a = 0 \wedge \nu d = \nu\kappa \wedge \nu b = 1 \wedge \nu c_r = msg(\nu d, \nu b) \wedge \nu\tau = \nu c_r\}$

then

$\mathbf{FES}(A) = \{\nu d = \nu\kappa \wedge \nu b = 1 \wedge \nu c_r = msg(\nu d, \nu b), \nu\tau = \nu c_r\}$

In other words, the $\mathbf{FES}()$ function removes those predicate terms that are irrelevant to the external events. It can be shown that $\mathbf{FES}(A \cup B) = \mathbf{FES}(A) \cup \mathbf{FES}(B)$ and $\mathbf{FES}(A \cap B) = \mathbf{FES}(A) \cap \mathbf{FES}(B)$.

4.2 Test Cases

A *mutant* is a specification of a fault model. For example, a mutant of Figure 1 is shown in Figure 4, in which the faulty transition T_2 points to a wrong state S_1.

A *test scenario* is a scenario whose external event can distinguish a correct EFSM from its mutants. That is, the external event sequence generated by the test scenario cannot be reproduced by the mutants. A test scenario p that distinguishes a correct EFSM M from a mutant F can be defined as follows:

Definition 3 *A scenario p is a test scenario for mutant F if $p \in (\beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!]))$.*

A *test case* is a prefix of the external event sequence generated by a test scenario. If h is a test scenario for an EFSM M and its mutant F, a test case is defined as follows:

Definition 4 *An event sequence T is a test case if*
1. $T = \langle h(\tau), h(\nu\tau), h(\nu^2\tau), ..., h(\nu^n\tau) \rangle$, *and*
2. $\forall g \in \mathbf{FES}(\beta[\![F]\!]), \exists i \leq n, [h(\nu^i\tau) \neq g(\nu^i\tau)]$.

The definition means that, a test case is a finite external event sequence that contains at least one element that cannot be reproduced by a faulty implementation. Therefore, one can recognize an incorrect implementation once an unexpected external event is produced.

4.3 Test Case Generation for Single-Module Specifications

For any EFSM M and its mutant F, it can be proved that $\beta[s] - \mathbf{FES}(\beta[\![F]\!]) \subseteq \beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!])$, where s is a state of M. Though a test scenario can be found anywhere

in $\beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!])$, it is more convenient to select the initial state as state s and to search for a test scenario in $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$. Finding a test scenario from the latter not only avoids setting up the implementation under test (IUT) to the starting state of the test scenario, but also reveals the correctness of the initial transition.

Let h be a scenario starting from state s. Let u be a state that the corresponding path of h passes through. Then, it can be shown that $h \in A \cap \beta[u]\,|^V$, for some assertion A and new version set V. Hence, any set of scenarios, H, can be represented as $\bigcup_h (A_h \cap \beta[u_h]|^{V_h})$, where h is a scenario in H, and A_h, u_h, and V_h are an assertion, a state, and a set of new versions, respectively. Let H and G be two sets of scenarios for EFSM M_h and M_g, respectively, such that $H = \bigcup_h (A_h \cap \beta[u_h]|^{V_h})$ and $G = \bigcup_g (A_g \cap \beta[u_g]\,|^{V_g})$. Then, $H - \mathbf{FES}(G)$ can be derived as follows:

$$H - \mathbf{FES}(G) \supseteq \bigcup_h \bigcap_g K_{h,g}$$

where $K_{h,g} =$

$$\begin{cases} A_h \cap \beta[u_h]|^{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) = \emptyset \quad (1a) \\ (A_h - \mathbf{FES}(A_g)) \cap \beta[u_h]|^{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) \neq \emptyset \wedge \\ & A_h - \mathbf{FES}(A_g) \neq \emptyset \quad (1b) \\ A_h \cap (\beta[u_h]|^{V_h} - \mathbf{FES}(\beta[u_g]|^{V_g})) & \text{otherwise} \quad (2) \end{cases}$$

As shown in the equations above, $K_{h,g}$ can be divided into Cases 1a, 1b, and 2, which are illustrated in Figure 5. Each circle in the figure represents a set of scenarios. In Case 2 (Figure 5a), A_h is covered by $\mathbf{FES}(A_g)$, meaning that for any scenario in A_h, there is a scenario in A_g that generates the same external event sequence as A_h does. Therefore, A_h along cannot distinguish H from G. To find the differences between H and G, $\beta[u_h]\,|^{V_h} - \mathbf{FES}(\beta[u_g]\,|^{V_g})$ is recursively calculated. In Case 1b (Figure 5b), some scenarios in A_h are not covered by $\mathbf{FES}(A_g)$, which means that a scenario that distinguishes H from G can be found in A_h. Therefore, $\beta[u_h]\,|^{V_h}$ is expanded to impose more restrictions to find a scenario in the shaded area. In Case 1a (Figure 5c), no scenarios in A_h are covered by $\mathbf{FES}(A_g)$, which means every scenario in A_h can be used to distinguish H from G. Therefore, the external events generated by a scenario in A_h can be used as a test case.

Let $\beta[s]$ be H, and $\beta[\![F]\!]$ be G. Using the result of the equations above, $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$ can be computed by the following algorithm.

1. Let there be a quintuple $\langle u, p, u', p', c\rangle$, where u and u' are assertions, p and p' are states, and c is 1a, 1b, or 2 with respect to Cases 1a, 1b, or 2 above. Let Q be a queue that initially contains $\langle s, true, u', true, 2\rangle$ for every state u' in F.
2. Get an element $\langle u, p, u', p', c\rangle$ from Q and check the following conditions.
   (a) If c is in Case 1a, check if there is any element, say $\langle v, q, v', q', d\rangle$ in Q such that $p = q$. If there is none, the corresponding path of assertion p generates a test case. Exit the program and return p.
   (b) If c is in Case 1b, for every outgoing transition t from state u, put $\langle v_t, (p \wedge p_t), u', p', d\rangle$ into Q, where v_t is the state t pointing to, p_t is the assertion imposed by the action of t, and d is Case 1a, 1b, or 2, based on the relation between $\{p \wedge p_t\}$ and $\mathbf{FES}(\{p'\})$.
   (c) If c is in Case 2, for every outgoing transition t from state u, and for every outgoing transition t' from state u', put $\langle v_t, (p \wedge p_t), v'_t, (p' \wedge p'_t), d\rangle$ into Q, where v_t and v'_t are the states t and t' pointing to, respectively; p_t and p'_t are the assertions imposed by the actions of t and t', respectively; and d is Case 1a, 1b, or 2, based on the relation between $\{p \wedge p_t\}$ and $\mathbf{FES}(\{p' \wedge p'_t\})$.
3. Repeat Step 2.

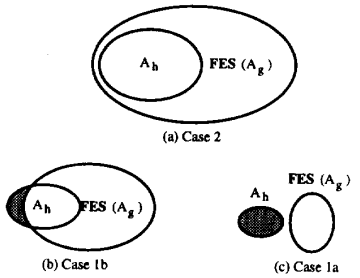


Figure 5: Illustration of the three cases of $K_{h,g}$

The idea of the algorithm can also be illustrated by Figure 5. Initially, every element in Q belongs to Case 2, which is Figure 5a. When $\beta[u]$ and $\beta[u']$ are expanded, the algorithm imposes more restrictions and less scenarios will satisfy the new restrictions. This makes the circles in Figure 5a "shrink." Eventually, the figure will become either Figure 5b or Figure 5c. If it is Figure 5c, a test scenario is found. Otherwise, the algorithm simply expand $\beta[u]$, such that the A_h circle in Figure 5b shrinks toward the shaded area and becomes Figure 5c.

Basically, it is an unsolvable problem as to whether two Turing-equivalent machines generate the same output. However, it is reasonable to assume that each transition can be finished in a bounded amount of time once it started, so that a limit can be set to the maximum number of expansions that can be performed. If the number of expansions exceeds this limit, and still no test cases can be found, we simply dictate that such faults are untestable. Noting that finding a test case for FSM has been proven by Yannakakis and Lee to be PSPACE-complete [20], we expect finding a test case for EFSM is at least as hard as the former. That is, there is unlikely any efficient algorithm besides exhaustive search. Due to the intractable nature of this problem, one can only rely on heuristic approaches to improve the performance. Some useful rules-of-thumb for the heuristic approach are discussed in Section 4.5.

4.4 Test Case Generation for Multiple Modules

Estelle allows several modules to be executed in parallel, communicating with each other through internal interaction points. Therefore, a specification of multiple modules describes a system containing several communicating EFSMs. To test such a system, the behavior of the system is defined as follows:

Definition 5 *If there are n modules, $M_1, M_2, ..., M_n$, in a specification, and let s_i be a state in module M_i, then*

$$\beta[s_1, s_2, ..., s_n] = \beta[s_1] \cap \beta[s_2] \cap \cdots \cap \beta[s_n]$$

Definition 6 *Let M be a specification that contains modules M_1, M_2, ..., M_n. Then,*

$$\beta[\![M]\!] = \bigcup_{s_1} \bigcup_{s_2} \cdots \bigcup_{s_n} \beta[s_1, s_2, ..., s_n]$$

where s_i is a state in M_i.

The behavior of a multiple-module system can be derived in the same way as the behavior of a single EFSM, except that every module has its own auxiliary variables τ and κ. It is assumed that different modules do not share the same variable names. Such a name conflict can easily be resolved by appending module names to the variable names. Figure 6 is a simple send-and-receive protocol, and its behavior is shown as follows:

Two scenarios for the sender and the receiver of the protocol can be derived as

$$\begin{aligned}
\beta[S] = \; & \{(\nu c_s = send(\nu d)) \wedge (\nu d = \nu\kappa_s) \wedge (\nu\tau_s = \nu c_s) \wedge \\
& (\nu c_i = msg(\nu d)) \wedge (\nu^2\tau_s = \nu c_i)\} \cap \\
& \beta[S]|^{\{\nu c_s, \nu d, \nu\kappa_s, \nu^2\tau_s, \nu c_i\}} \\
\beta[R] = \; & \{(\nu c_i = msg(\nu b)) \wedge (\nu b = \nu\kappa_r) \wedge (\nu\tau_r = \nu c_i) \wedge \\
& (\nu c_r = rec(\nu b)) \wedge (\nu^2\tau_r = \nu c_r)\} \cap \\
& \beta[R]|^{\{\nu c_r, \nu b, \nu\kappa_r, \nu^2\tau_r, \nu c_i\}}
\end{aligned}$$

Therefore,

$$\begin{aligned}
& \beta[S,R] = \beta[S] \cap \beta[R] = \\
& \{(\nu\tau_s = \nu c_s = send(\nu d)) \wedge (\nu d = \nu\kappa_s)\} \cap \\
& \{(\nu^2\tau_s = msg(\nu d) = \nu c_i = msg(\nu b) = \nu\tau_r) \wedge (\nu b = \nu\kappa_r)\} \cap \\
& \{(\nu^2\tau_r = \nu c_r = rec(\nu b))\} \cap \\
& \beta[S]|^{\{\nu c_s, \nu d, \nu\kappa_s, \nu^2\tau_s, \nu c_i\}} \cap \beta[R]|^{\{\nu c_r, \nu b, \nu\kappa_r, \nu^2\tau_r, \nu c_i\}}
\end{aligned}$$

Note that channel c_i relates the output of the sender to the input of the receiver. Since $\nu c_i = msg(\nu d)$ in the sender and $\nu c_i = msg(\nu b)$ in the receiver, νd equals to νb. Note also that each module has its own variables τ and κ. The external events are those τ's which record events on the channels that connect external IPs. For example, $\nu\tau_s$ and $\nu^2\tau_r$ are external events since $\nu\tau_s = \nu c_s$, $\nu^2\tau_r = \nu c_r$, and c_s and c_r are connected to external IPs. The order of external events is preserved by variables τ's. For instance, because $\nu\tau_s$ and $\nu\tau_r$ precede $\nu^2\tau_s$ and $\nu^2\tau_r$, respectively, and because $\nu^2\tau_s = \nu\tau_r$, $\nu\tau_s$ precedes $\nu^2\tau_r$.

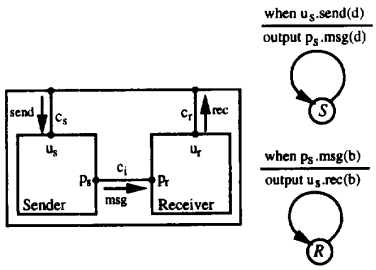


Figure 6: A simple send-and-receive protocol

The mutants of a multi-module system can be specified by Estelle, and their behavior can also be described by the behavior model. Hence, the test case for a multi-module specification can be generated using the same algorithm for a single module specification.

Estelle allows for dynamic creating and destroying of a module instance through statements **init** and **release**, respectively. It can also dynamically create and destroy a channel between two IPs by issuing **connect, disconnect, attach**, and **detach** statements. Therefore, an unique name must be assigned to each newly created object. For example, if statement "**init** mv **to** *body*" is issued (where mv is a "module variable" and *body* is a "module body"), a variable x in module body "*body*" should be referred to as $mv.n.x$, where n is the number of times mv is initialized to a module body. If statement "**connect** p **to** q" is issued (where p and q are IPs), a new variable for the channel between p and q is created with a name such as $C_{p,q}$. With the assurance that each object has its own name, the algorithm in Section 4.3 can run without a hitch.

4.5 Heuristic Search Methods

Since the conformance of two Turing-equivalent machines is generally an unsolvable problem, it is unlikely to find a test case generation algorithm without using an exhaustive search. However, heuristics can be used to improve the performance of the test case searching process.

From our experience, there are some guidelines that can be used to improve the performance of the algorithm in Section 4 dramatically:

1. Expand those items in queue Q that lead to faulty transitions first.
2. After executing a faulty transition, traverse through those paths which output the values of the variables that appear in the faulty transition first.
3. If there are more than one path in the above, traverse the shortest one first.

Using these guidelines, the algorithm described in Section 4.3 can be modified as follows:

1. Let Q be a priority queue.
2. For every element $\langle u, p, u', p', c \rangle$ that is to be added to Q, check the following:

(a) If the corresponding path, say P', of predicate p' does not contain any faulty transition, set the priority value of the element to be the length of the shortest path from the last transition of P' to a faulty transition in F.

(b) If the corresponding path P' of predicate p' contains a faulty transition, set the priority value of the element to be the length of the shortest path between the last transition in P' and a transition that refers to any variable used in the the faulty transition.

3. To select an element in Q, choose the one with the lowest priority value.

5 Conclusion

In this paper, an automatic test case generation method for an ISO standard specification language, Estelle, is presented. The method compares the behavior of a specification to the behavior of a given fault model. Based on their difference, test cases are mechanically derived. Unlike other conformance testing methods that apply only to fixed fault models, the proposed method is able to generate a test case that detects possible implementation errors specified by a given fault model. Therefore, one can test those critical faults and obtains confidence in the coverage of such faults. When time is critical and resources are limited, it becomes very important to be able to test the most critical faults and frequently executed transitions first.

While most of the existing methods concern only generating test cases for a single entity, the proposed method is able to deal with those specifications that contain multiple modules. Treating channels between two modules as variables, the method transforms the I/O statements into assignment statements and derives test cases by using the same algorithm used to generate test cases for a single module.

Test case generation has been proven to be at least PSPACE-hard. To improve the performance of the proposed algorithm, heuristics are introduced. The guidelines suggested in this paper significantly reduce the number of states exploited, thereby improving the performance of the test case generation processes.

In summary, the method proposed in [16] is extended to generating test cases for Estelle. It is also extended to dealing with multiple modules. In addition, some heuristics are suggested to improve the performance of the test case generation process. Currently, a test case generator for EFSM with given fault models has been developed, and an extension of it to Estelle is under consideration.

References

[1] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, March 1978.

[2] G. Gönenç, "A model for the design of fault detection experiments," *IEEE Trans. on Computers*, vol. C-19, no. 6, pp. 551–558, June 1970.

[3] S. Naito, "Fault detection for sequential machines by transition tours," in *Proc. 11th IEEE Symp. on Fault Tolerant Computing*, pp. 238–243, 1981.

[4] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks and ISDN Systems*, vol. 15, pp. 285–297, 1988.

[5] G. v. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols," in *Proc. IFIP Congress '77*, pp. 229–234, 1977.

[6] S. Budkowski and P. Dembinski, "An introduction to Estelle: A specification language for distributed systems," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 3–23, 1987.

[7] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1987.

[8] E. Brinksma, "A theory for the derivation of tests," in *The Formal Description Technique LOTOS* (P. van Eijk, C.A.Vissers, and M. Diaz, eds.), pp. 235–247, Elsevier Science Publishers B.V. (North-Holland), 1989.

[9] R. Langerak, "A testing theory for LOTOS using deadlock detection," in *Proc. 10th IFIP Symp. on Protocol Specification, Testing, and Verification*, pp. 87–98, 1990.

[10] B. Sarikaya, G. V. Bochmann, and E. Cerny, "A test design methodology for protocol testing," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 5, pp. 518–531, May 1987.

[11] H. Ural, "Test sequence selection based on static data flow analysis," *Computer Communications*, vol. 10, no. 5, pp. 234–242, 1987.

[12] H. Ural and B. Yang, "A test sequence selection method for protocol testing," *IEEE Trans. on Communications*, vol. 39, no. 4, pp. 514–523, April 1991.

[13] C.-J. Wang and M. T. Liu, "Axiomatic test sequence generation for extended finite state machines," in *Proc. 12th International Conference on Distributed Computing Systems*, pp. 252–259, June 1992.

[14] C.-J. Wang and M. T. Liu, "A test suite generation method for extended finite state machines using axiomatic semantics approach," in *IFIP Trans. Protocol Specification, Testing, and Verification, XII*, pp. 29–43, North-Holland, 1992.

[15] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, "Fault models in testing," in *Protocol Test Systems, IV*, pp. 17–30, Elsevier Science Publisher B.V. (North-Holland), 1992.

[16] C.-J. Wang and M. T. Liu, "Generating test cases for EFSM with given fault models," in *Proc. IEEE INFOCOM '93*, pp. 774–781, March 1993.

[17] L. A. Clarke and D. J. Richardson, "Applications of symbolic evaluation," *Journal of Systems and Software*, vol. 5, pp. 15–35, 1985.

[18] J. C. King, "Symbolic execution and program testing," *CACM*, vol. 19, no. 7, pp. 385–394, July 1979.

[19] F. G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer*, ch. 4, pp. 193–215. Prentice-Hall, Inc., 1981.

[20] G. J. Holzmann, *Design and Validation of Computer Protocols*, ch. 9, pp. 198–199. Prentice-Hall, Inc., 1991.