PROTOCOL CONFORMANCE USING A PROGRESSIVE TEST APPROACH

Jorge Mamede
INESC Porto / I.S.E.P / F.E.U.P.
Campus da FEUP, Rua Dr. Roberto Frias,
Porto, Portugal
email: jmamede@inescporto.pt

Eurico Carrapatoso and Manuel Ricardo
INESC Porto / F.E.U.P.
Campus da FEUP, Rua Dr. Roberto Frias,
Porto, Portugal
email: {emc,mricardo}@inescporto.pt

ABSTRACT
The development of communications systems demands testing. This paper presents a framework for testing on-the-fly, which relies on the definition of 3 types of tests and on their sequential execution. The *ioco* conformance relation was considered in order to assign verdicts.

A tool prototype is also presented that supports the proposed framework. This tool, named PROFYT, was developed based on the SPIN verifier, and uses communicating FSMs to describe the specification. The test of Conference Protocol implementations was carried out on-the-fly with PROFYT and enabled us to conclude about the benefits of the test methodology proposed.

KEY WORDS
On-the-fly conformance testing, PROFYT.

1 Introduction

The development of communications systems demands testing. During decades, many testing methodologies were defined aimed at verifying the conformity of protocol implementations to their specifications [1], [2], [3], [4], [5], [7], [8], [9]. One of the recent approaches consists of testing *on-the-fly* the implementation conformity [11]. Tools implementing the *on-the-fly* conformance testing approach, derive and execute tests in a single step; relying on some specification model, these tools explore the model not only to select the next message to be transmitted by the tester, but also to validate messages received by the implementation. Although appealing, this testing approach has some drawbacks such as (1) the length of test traces and (2) the difficulty in reproducing detected failures [11]. Besides, the lack of test control may conduce to situations in which sever non-conformance cases are undetected, just because the test events selected randomly did not exercise some basic interoperability function.

In this paper we present a new methodology and tool which uses the on-the-fly approach, but reduces the problems mentioned. Different test types were defined, which address different type of faults, meaningful for the implementer. The complete test execution process demands the execution of all the test types, and the *ioco* [10] conformance relation was adopted to assign verdicts. The tool implementing the method, named PROFYT, is based on the SPIN [6] verifier and uses communicating FSMs as behaviour model. The validation of the method and the tool is based on the Conference Protocol; multiple faulty implementations were tested, and the cost of finding faults is compared with the TorX [11] tool. The results obtained enabled us to conclude and quantify the benefits of the proposed approach.

This work is reported in 6 sections. Section 2 defines the communicating finite state machines. Section 3 presents the main contribution of this paper: the progressive test method, the test modes, and the algorithms implementing them. Section 4 presents the PROFYT tool, which implements the methodology proposed. Section 5 reports the results of evaluating our methodology and tool against the TorX tool. Section 6 concludes the paper.

2 Communicating Finite State Machines

Communicating finite state machines are used to describe the behaviour of interacting processes [8], and can be extended with message queues and variables [6].

A message queue m is a triple $m = (\,U_m,\, N_m,\, C_m\,)$, where U_m is the set of messages, N_m is the maximum number of messages held by the queue, and C_m is the set of ordered sets of messages held by the queue; M denotes the set of queues used by a state machine, and $m \in M$. An ordered set of messages $c_m =< u_{m,1}, ..., u_{m,i}, ..., u_{m,k} >$ is an element of C_m, where $u_{m,i} \in U_m$ is the message occupying the i^{th} position of queue m, and $1 \leq k \leq N_m$; $c_m =<>$ represents the queue m when it is empty, and $\#c_m$ the number of messages held by queue m. The state of a variable l is denoted by v_l and its initial state is denoted by v_{l_0}; the state of the x machine variables is jointly represented by $w = (v_1, ..., v_l, ..., v_x)$, being W the set of all possible w, and $w \in W$. Q is the finite set of state machine control states. The state machine global space state is then given by $G = Q \times C \times W$, and contains states $g_i = (q_i, (c_{1_i}, ..., c_{m_i}), (v_{1_i}, ..., v_{l_i}, ..., v_{x_i}))$, where $g_i \in G$.

An Extended Finite State Machine is defined by $P = (G_P, g_{0_P}, A_P, T_P, M_P)$. G_P is the finite and non empty set of global states; g_{0_P} is the initial state; A_P is the set of actions of P; $T_P \subseteq G_P \times A_P \times G_P$ is the transition relation of P; M_P is the set of message queues used by P.

The P actions are given by $A_P = \mathcal{I}_P \cup \mathcal{O}_P \cup \mathcal{W}_P \cup$

$\{\tau\}$. $\mathcal{I}_P$ is the set of input symbols of P, representing the reception of messages from the queues. $\mathcal{O}_P$ is the set of output symbols, representing the transmission of messages to queues, and $\mathcal{I}_P \cap \mathcal{O}_P = \emptyset$. $\mathcal{W}_P$ is the set of symbols denoting the operations over the machine variables. τ labels the transitions between states with no execution of actions in $\mathcal{I}_P$, $\mathcal{O}_P$, or $\mathcal{W}_P$. The execution of actions in $\mathcal{I}_P$ or $\mathcal{O}_P$ depends on the state of the message queues.

A transition $t \in T_P$ results from the execution of an action $a \in A_P$ and, leads the machine from the state g_{i_P} to state g_{i+1_P}. This transition is represented by $t(\,g_{i_P}, a\,) = g_{i+1_P}$, or $(g_{i_P}\,,\,a\,, g_{i+1_P}) \in T_P$.

For each action $a \in \mathcal{I}_P \cup \mathcal{O}_P$, a unique $d(a)$ identifies the queue used by the action a; the message transferred through the $d(a)$ queue is represented by $msg(\,a\,)$. As N_m represents the number of slots in the message queue m, $N_m > 0$, a transition resulting from the execution of action $a \in \mathcal{I}_P$ is possible or executable when $d(\,a\,)$ queue is not empty, i.e. $\#c_{d(a)} \neq 0$ and $msg(\,a\,) = u_{d(a),1}$. In state g_{i+1_P}, after the execution of a, the state of queue $c_{d(a)_{i+1}}$ is defined by $c_{d(a)_{i+1}} = c_{d(a)_i} \setminus \{u_{d(a),1}\}$. A transition resulting from an action $a \in \mathcal{O}_P$ is executable when $d(a)$ queue is not full, i.e. $\#c_{d(a)} < N_{d(a)}$ and $msg(a) \in U_{d(a)}$. In this case the new state of queue $d(a)$ is defined by $c_{d(a)_{i+1}} = c_{d(a)_i} \cup \{u_{d(a),k}\}$, where $u_{d(a),k} = msg(a)$ and $k = \#c_{d(a)_i} + 1$.

A quiescent state g_δ is a state having only outgoing transitions labelled with input actions. The set of quiescent states of machine P is defined by:

$$\Delta_P = \{\, g_\delta \in G_P \mid \forall_{\,a \in (A_P \setminus \mathcal{I}_P)} \;:\; t(\,g_\delta,\, a\,) \notin G_P \,\}$$

A_P^* represents the set of all the ordered combinations of A_P actions. A *trace* is an ordered set of actions executed by P, and it is given by $\sigma \in A_P^*$. The length of the σ trace is respresented by $|\,\sigma\,|$. The concatenation of two traces σ_a and σ_b is represented by $\sigma_a.\sigma_b$, while $\sigma_c.a$ denotes the concatenation of trace σ_c with the action a. Moreover, a function $tail(\sigma)$ identifies the last action of σ, such that $tail(\sigma_a.a) = a$. The function $head(\sigma)$ identifies the first action of σ, such that $head(a.\sigma_b) = a$.

Moving from a state by executing a trace leads to extended transitions $\widehat{T}_P \subseteq G_P \times A_P^* \times G_P$, represented by $\hat{t}(\,g,\,\sigma\,) = g'$ or $(\,g,\,\sigma,\,g'\,) \in \widehat{T}_P$. The set of all the traces defined in P is represented by $traces(P)$.

The composition of machines $X = (G_X, g_{0_X}, A_X, T_X, M_X)$ and $Y = (G_Y,\, g_{0_Y},\, A_Y,\, T_Y,\, M_Y)$ is defined by a machine $Z = (G_Z,\, g_{0_Z},\, A_Z,\, T_Z,\, M_Z)$ where $G_Z = G_X \times G_Y$. (Since $G_Z = Q_Z \times C_Z \times W_Z$, now $Q_Z = Q_X \times Q_Y$, $C_Z = C_X \times C_Y$, and $W_Z = W_X \times W_Y$); g_{0_Z} is the initial state, defined by $g_{0_Z} = (g_{0_X},\, g_{0_Y})$; $A_Z = A_X \cup A_Y = \mathcal{I}_Z \cup \mathcal{O}_Z \cup \mathcal{W}_Z \cup \{\tau\}$ is the set of actions of Z, in which $\mathcal{I}_Z = \mathcal{I}_X \cup \mathcal{I}_Y$, $\mathcal{O}_Z = \mathcal{O}_X \cup \mathcal{O}_Y$ and $\mathcal{I}_Z \cap \mathcal{O}_Z = \emptyset$. $\mathcal{W}_Z = \mathcal{W}_X \cup \mathcal{W}_Y$ is the set of symbols representing the manipulation of Z variables; $T_Z \subseteq G_Z \times A_Z \times G_Z$ defines the transitions between states of Z; $M_Z = M_X \cup M_Y$ is the set of message queues of Z.

Communication through message queue m can be classified as asynchronous or synchronous. The communication is asynchronous when the queue has a non-null number of message slots, $N_m > 0$. In this case a transition $((g_{i_X}, g_{i_Y}),\, a,\, (g_{i+1_X}, g_{i+1_Y}))$ is defined in T_Z if either: **1.** $\forall\, a \in \mathcal{I}_X \cup \mathcal{O}_X, (g_{i_X}, a, g_{i+1_X}) \in T_X, \exists g_{i_Y}, g_{i+1_Y} \in G_Y : (d(a) \in M_Y \wedge (g_{i_Y}, \tau, g_{i+1_Y}) \in T_Y) \vee (d(a) \notin M_Y \wedge g_{i_Y} = g_{i+1_Y})$; or **2.** $\forall\, a \in \mathcal{I}_Y \cup \mathcal{O}_Y, (g_{i_Y}, a, g_{i+1_Y}) \in T_Y, \exists g_{i_X}, g_{i+1_X} \in G_X : (d(a) \in M_X \wedge (g_{i_X}, \tau, g_{i+1_X}) \in T_X) \vee (d(a) \notin M_X \wedge g_{i_X} = g_{i+1_X})$; or **3.** $\forall\, a \in A_X \setminus (\mathcal{I}_X \cup \mathcal{O}_X) \wedge (g_{i_X}, a, g_{i+1_X}) \in T_X, \exists g_{i_Y}, g_{i+1_Y} \in G_Y : g_{i_Y} = g_{i+1_Y}$; or **4.** $\forall\, a \in A_Y \setminus (\mathcal{I}_Y \cup \mathcal{O}_Y) \wedge (g_{i_Y}, a, g_{i+1_Y}) \in T_Y, \exists\, g_{i_X}, g_{i+1_X} \in G_X : g_{i_X} = g_{i+1_X}$.

The communication is synchronous when the queue has a zero message capacity $(N_m = 0)$ and, therefore, a transmission through this queue requires that the receiver is able to simultaneously accept the message. In this case, the transition $((g_{i_X}, g_{i_Y}), a, (g_{i+1_X}, g_{i+1_Y}))$ is defined in T_Z by replacing the conditions **1.** and **2.** mentioned above by **1.** $\forall a \in \mathcal{O}_X, b \in \mathcal{I}_Y, \exists\, (g_{i_X}, a, g_{i+1_X}) \in T_X, (g_{i_Y}, b, g_{i+1_Y}) \in T_Y : d(a) = d(b) \wedge msg(a) = msg(b)$; or **2.** $\forall a \in \mathcal{I}_X, b \in \mathcal{O}_Y, \exists\, (g_{i_X}, a, g_{i+1_X}) \in T_X, (g_{i_Y}, b, g_{i+1_Y}) \in T_Y : d(a) = d(b) \wedge msg(a) = msg(b)$.

The set of Z states resulting from the composition of quiescent states of X is

$$\Delta_Z^X = \{\, g \in G_Z \mid \forall_{\,g_\delta \in \Delta_X,\; g_y \in G_Y} \;:\; g = (g_\delta,\, g_y)\,\}$$

3 Progressive Conformance Method

Let us consider a protocol specified by a set of communicating extended finite state machines. After composition, the specification is assumed to be represented by $S = (\,G_S,\, g_{0_S},\, A_S,\, T_S,\, M_S\,)$.

The architectural and functional characteristics of the tester depend strongly on the specification model. S is said to be an open model in the sense that the behaviour of its environment is not described. In order to generate tests, a "maximum behaviour environment" needs to be created which closes the open model S. This environment is described by a machine that can always send and receive all the messages; thus, it can generate every sequence of inputs of S, and receive every output sequence generated by S. This environment is represented by the state machine $E = (\,G_E,\, g_{0_E},\, A_E,\, T_E,\, M_E)$. The actions of A_E are either message transmissions or receptions, and are related with the transmissions and receptions of S. The set $\mathcal{O}_E$ is defined by $\mathcal{O}_E = \{a' | a \in \mathcal{I}_\mathcal{S} \wedge msg(a) = msg(a') \wedge a' \notin \mathcal{O}_\mathcal{S}\}$ and the set $\mathcal{I}_E$ of reception actions is given by $\mathcal{I}_E = \{a' | a \in \mathcal{O}_\mathcal{S} \wedge msg(a) = msg(a') \wedge a' \notin \mathcal{I}_\mathcal{S}\}$. The transitions of E satisfy the condition $\forall a \in A_E : (g_{0_E}, a, g_{0_E}) \in T_E$. The set M_E is given by $M_E = \{d(a) | a \in \mathcal{I}_E \cup \mathcal{O}_E\}$.

When S is composed with the specification E, a closed machine is obtained. This machine, named *closed specification* (**C**), represents the composition of state machines S and E, and it is described by $\mathbf{C} = (G_{\mathbf{C}}, g_{0_{\mathbf{C}}}, A_{\mathbf{C}}, T_{\mathbf{C}}, M_{\mathbf{C}})$

The behaviour of our tester is inferred from $\mathbf{C}$. The architecture of the tester is imposed by the queues of E. The tester actions are defined by the actions of E. The test transitions are obtained by exploring $\mathbf{C}$; the reception of a message by the tester is possible only if the reception of the message is also possible on $\mathbf{C}$.

The tester $\mathbf{T}$ can be described by $\mathbf{T} = (G_{\mathbf{T}}, g_{0_{\mathbf{T}}}, A_{\mathbf{T}}, T_{\mathbf{T}}, M_{\mathbf{T}})$ where $G_{\mathbf{T}} = (Q_{\mathbf{T}} \times C_{\mathbf{T}} \times W_{\mathbf{T}}) \cup \{\mathbf{pass}, \mathbf{fail}\}$ is the set of $\mathbf{T}$ states; $g_{0_{\mathbf{T}}} = g_{0_{\mathbf{C}}}$ is the initial state of $\mathbf{T}$; $A_{\mathbf{T}} = \mathcal{I}_{\mathbf{T}} \cup \mathcal{O}_{\mathbf{T}} \cup \mathcal{W}_{\mathbf{T}} \cup \{\tau\}$ is the actions set; $T_{\mathbf{T}} \subseteq G_{\mathbf{T}} \times A_{\mathbf{T}} \times G_{\mathbf{T}}$ is the set of $\mathbf{T}$ transitions; $M_{\mathbf{T}}$ is the set of $\mathbf{T}$ queues. $\mathcal{O}_{\mathbf{T}}$ represent the actions of $\mathcal{O}_E$. $\mathcal{I}_{\mathbf{T}}$ includes also two additional input actions, $\mathcal{I}_{\mathbf{T}} = \mathcal{I}_E \cup \{\xi, \delta'\}$; ξ represents the reception of unknown messages; δ' represents the detection of an invalid quiescence state on the implementation under test (iut). The queues $M_{\mathbf{T}}$ are replicas of the queues M_E; however, the vocabulary of $M_{\mathbf{T}}$ queues is larger than the vocabulary of M_E queues, in order to accommodate the invalid *iut* messages. The $G_{\mathbf{T}}$ and $T_{\mathbf{T}}$ sets are defined dynamically by executing simultaneously the tester and the iut. Let us consider that the iut is modelled by the, a priori unknown, model $I = (G_I, g_{0_I}, A_I, T_I, M_I)$, and assume that $M_I = M_{\mathbf{T}}$, in order to enable the interoperation between I and $\mathbf{T}$.

Initially, we consider that the tester $\mathbf{T}$ has an empty set of transitions, and it is in its initial state $g_{0_{\mathbf{T}}}$. During the test execution, messages are exchanged between $\mathbf{T}$ and I through the $M_{\mathbf{T}}$ queues. Testing is realised by checking the queues $M_{\mathbf{T}}$ for messages sent by I. When, according to specification $\mathcal{S}$, the iut has no messages to send, we say that the iut is in a quiescent state. In this case, $\mathbf{T}$ is required to transmit a message, being each transmission of $\mathbf{T}$ preceded by a message selection phase on $\mathbf{C}$. The transitions of $\mathbf{T}$ are defined by the routine *RunTest*( ) :

```
T_T = { }                                        /* set of T transitions */
t_c ∈ T_C ; t_t ∈ T_T
g_t ∈ G_T ; g_c ∈ G_C
QueuesWithMessages = { }        /* set of queues having messages from the iut */

RunTest( )
{
.   g_t = g0_T ; g_c = g0_C
.   m ∈ QueuesWithMessages
.   GetQueuesWithMessages( )              /* updates the set QueuesWithMessages*/
.   while g_t ≠ fail
.   {
.   .   if #QueuesWithMessages == 0
.   .   {
.   .   .   if IsQuiescent( g_c )             /* verifies if quiescence is valid, */
.   .   .   {                                 /* i.e. r ∈ I_E is unreachable from state g_c */
.   .   .   .   if ( SortNextTxMsg( g_c ) )  /* selects randomly the next O_E action reachable*/
.   .   .   .   {                             /* from state g_c, and obtains the σ trace to the action */
.   .   .   .   .   G_T = G_T ∪ { g'_t } /* creates a new test state g'_t in G_T and defines*/
.   .   .   .   .                          /* a new T_T transition to it on the execution */
.   .   .   .   .   T_T = T_T ∪ { ( g_t , tail( σ ), g'_t) /* of the action tail(σ) */
.   .   .   .   .   g_t = t̂_T(g_t , tail( σ ) )
.   .   .   .   .   g_c = t̂_C( g_c , σ )   /* updates the states of T and C */
.   .   .   .   .   SendMsg( tail( σ ) )   /* transmits the msg(tail(σ)) action to iut*/
.   .   .   .   .   GetQueuesWithMessages( )
.   .   .   .   }
.   .   .   }
.   .   .   else                           /* if invalid quiescence is detected */
.   .   .   {
.   .   .   .   T_T = T_T ∪ { ( g_t , δ' , fail ) }   /* a transition with δ' action is added to */
.   .   .   .   g_t = t_T(g_t , δ' ) = fail          /* T leading it to the fail state*/
.   .   .   }
.   .   }
.   .   else                          /* #QueuesWithMessages ≠ 0 */
.   .   {
.   .   .   if RcvdMsgIsValid ( m, g_c )   /* validates the first message received in the queue m, */
.   .   .   {                               /* and obtains the σ trace toward an action of */
.   .   .                                   /* I_E labeling the reception of that message; */
.   .   .   .   G_T = G_T ∪ { g'_t }        /* creates a new test state g'_t in G_T, and defines */
.   .   .   .   .                           /* a new T_T transition to it on the execution */
.   .   .   .   T_T = T_T ∪ (g_t , tail( σ ), g'_t)     /* of the action tail(σ) */
.   .   .   .   g_t = t̂_T(g_t , tail( σ ) )
.   .   .   .   g_c = t̂_C( g_c , σ )                    /* updates the states of T and C */
.   .   .   .   RcvMsg( m )                              /* removes first message from queue m */

.   .   .   .   GetQueuesWithMessages( )
.   .   .   }
.   .   .   else                  /* if the received msg is not valid */
.   .   .   {
.   .   .   .   T_T = T_T ∪ { ( g_t , ξ, fail ) }    /* a transition with ξ action is added to */
.   .   .   .   g_t = t_T(g_t , ξ ) = fail           /* T leading it to the fail state*/
.   .   .   }
.   .   }
.   }
}
```

This function describes the steps leading to the definition of the $\mathbf{T}$ states set $G_{\mathbf{T}}$ and transitions set $T_{\mathbf{T}}$. It is based on the continuous monitoring for messages in the queues of $M_{\mathbf{T}}$. When there are not queues with messages, the implementation quiescence is evaluated on $\mathbf{C}$, and if it is valid, the tester initiates a random search for next action labelling the message to transmit; based on the identified action, a new transition is added to $\mathbf{T}$, and the corresponding message is transmitted to the iut. If the detected quiescence is not valid the test terminates in the **fail** state. When a message is received from iut in the queues of $M_{\mathbf{T}}$, a search is performed on $\mathbf{C}$ to check if it is valid, i.e. if there is a reachable action labeling the reception of that message; in this case, a new transition is added to $\mathbf{T}$ that is based on that action. If the received message is not valid the test terminates and a transition leading $\mathbf{T}$ toward the **fail** state is added.

The set $QueuesWithMessages$ is a list of queues containing messages to be consumed by $\mathbf{T}$; this set is defined by the function *GetQueuesWithMessages*( ), and it is rebuilt after the reception or transmission of every message. When the $QueuesWithMessages$ set is empty, the quiescence of I is verified in $\mathbf{C}$, by the function *IsQuiescent*( ), that searches $\mathbf{C}$ for an action $r \in \mathcal{I}_E$; a quiescence is valid if r is unreachable. If the quiescence observed is not specified, i.e. if there is at least one reachable $r \in \mathcal{I}_E$, a new transition leading the test to the **fail** state is added to $T_{\mathbf{T}}$. The action associated to this transition is δ' , denoting the invalid quiescence detection, and terminating the test. When the detected quiescence is valid, the test $\mathbf{T}$ selects a message to transmit to I. The selection of this message is performed by the function *SortNextTxMsg*( ), which searches the model $\mathbf{C}$ for an action in $\mathcal{O}_E$. The trace σ leading to the selected action is used to guide the $\mathbf{C}$ execution. The extended transition, from state g_c and using the actions of σ, is executed in order to refresh the state of $\mathbf{C}$. Based on the action selected, a new state is added to $G_{\mathbf{T}}$, and a transition to this state is added to $T_{\mathbf{T}}$. The tester proceeds with the execution of the new transition and the function *SendMsg*( ) is invoked for transmitting the corresponding message. After each test transmission, the M_T queues are verified again for new messages, and the $QueuesWithMessages$ is rebuilt.

The function *RcvdMsgIsValid*( ) is applied to the first message of queue $m \in QueuesWithMessages$, which searches for a σ trace in $\mathbf{C}$, guiding the machine toward an action of $\mathcal{I}_E$ labelling the reception of that message. If such trace exists, the message received is valid, and the test must proceed. Therefore, a new state is defined in $G_{\mathbf{T}}$, the action of $\mathcal{I}_{\mathbf{T}}$ labelling the reception of that message is selected and, using them, a new transition is created in $\mathbf{T}$ that accepts the message and leads the tester to a new state. The test continues with the execution of σ actions on the $\mathbf{C}$ machine and, by executing the *RcvMsg*(m) function, removes the message from queue the $m \in M_{\mathbf{T}}$. Then, the $QueuesWithMessages$ set is rebuilt, and the next message in queues is evaluated. If the σ trace becomes empty, the received message is invalid; it is not allowed by the specification and, therefore, could not be transmitted by I. In this case, the test enters in a **fail** state and the test terminates.

3.1 Optimised Test Modes

The random algorithm presented enables to test *on-the-fly* an *iut*; they exchange messages until a message is sent by the *iut* which is not allowed by the specification. The *ioco* conformance relation defined in [10] is adopted. When a fault is found, the test log enables its characterisation. After the fault is eliminated, a new test session shall be initiated, until some pre-defined criteria for ending the test is reached. This approach brings problems, such as, (1) the length of test traces and (2) the difficulty in reproducing detected failures. Besides, the lack of test control may conduce to situations in which sever non-conformance cases are undetected, just because the test events selected randomly did not exercise some basic interoperability function.

In order to alleviate these problems, three additional testing algorithms are proposed. These algorithms address 3 types of behaviour commonly observed during the test sessions by human operators: $1)$ a correct *iut* usually answers immediately to a received message; $2)$ a correct *iut* accepts messages leading to quiescent states and do not answer them; $3)$ a correct *iut* usually discards silently messages that are invalid or unexpected. We defined one testing algorithm for each of these commonly observed behaviours; each algorithm is associated to what we called a test mode.

3.1.1 Special traces and actions

The definition of these algorithms demands the characterisation of some special behaviour traces. The classification of traces is usually carried out after an *iut* reaches a quiescent state, and by simulating the possible behaviour paths through the reachability graph of S. These simulations are initiated at the quiescent state and explore all the inputs until one of the following conditions is detected: 1) an output action is detected, which corresponds to an input action of E; 2) a quiescent state is detected; 3) the maximum simulation depth is reached.

Traces are classified according to the condition that terminates the simulation. Let us consider that the execution of a test $\mathbf{T}$ leads the machine $\mathbf{C}$ to a state $g \in \Delta^S_{\mathbf{C}}$, and also t output actions in $\mathcal{O}_E$ matching all the implementation inputs specified for a state g. Each t action can initiate three classes of traces:

i) Ψ **traces** lead $\mathbf{C}$ to the input actions $r \in \mathcal{I}_E$; the set $\Psi(g,t)$ contains the Ψ traces initiated with the action t on state g. The t actions initiating the Ψ traces belong to the set A_ψ, defined by $A_\psi(g) = \{t \in \mathcal{O}_E | \exists_{g \in G_{\mathbf{C}}} : \Psi(g,t) \neq \emptyset\}$.

ii) Φ **traces** lead $\mathbf{C}$ to the quiescent states $g_\delta \in \Delta^S_{\mathbf{C}}$; the set $\Phi(g,t)$ contains the Φ traces initiated with the action t on state g. The actions $t \in \mathcal{O}_E$ initiating the Φ traces belong to the set A_ϕ, defined by $A_\phi(g) = \{t \in \mathcal{O}_E | \exists_{g \in G_{\mathbf{C}}} : \Phi(g,t) \neq \emptyset\}$.

iii) Γ **traces** have length 1 and do not change the state of $\mathbf{C}$, or lead $\mathbf{C}$ to quiescent states, for which no judgement is possible. The $\Gamma(g,t)$ set contains the Γ traces initiated with t on state g that either ignore t or that do not belong to $\Psi(g,t)$ nor $\Phi(g,t)$. The t actions of $\mathcal{O}_E$ starting the traces of $\Gamma(g,t)$, belong to the set $A_\gamma(g)$ defined by $A_\gamma(g) = \{t \in \mathcal{O}_E | \exists_{g \in G_{\mathbf{C}}} : \Gamma(g,t) \neq \emptyset\}$.

3.2 Test Mode_1

Test Mode_1 aims at detecting faults related to the first type of behaviour mentioned in Section 3.1. The cases of non conformance that can be detected using this test mode are invalid answering messages, missing messages, and incorrect message coding. The selection of test actions in this test mode is made by the function *SelectTM1TxMsg*, instead of the *SortNextTxMsg*, presented above.

```
σ ∈ traces(C)

Boolean SelectTM1TxMsg( g ∈ G_C )
{
.       σ = { }
.       ClassifyActions( g )                  /* build sets Ψ, Φ and Γ using actions of O_E on the state g */
.       if A_ψ(g) ≠ ∅      a ∈ A_ψ(g)                 /* random selection of action a from A_ψ(g) */
.       .   else if A_φ(g) ≠ ∅      a ∈ A_φ(g)        /* random selection of action a from A_φ(g) */
.       .   .   else if A_γ(g) ≠ ∅      a ∈ A_γ(g)    /* random selection of action a from A_γ(g) */
.       .   .   .   else return FALSE
.       FindAction( g, a, O_E )               /* define the trace σ from state g to the action a of O_E */
.       return TRUE
}
```

The *SelectTM1TxMsg* function starts with the classification of the test actions executable from state g, and their distribution by the sets A_ψ, A_ϕ or A_γ, according to the trace they initiate. Then, one test action a is randomly selected from these sets depending on their emptiness. At the end, the *FindAction* function is used to identify the σ trace, that will be used to drive the machine $\mathbf{C}$ toward the selected action a.

3.3 Test Mode_2

The Test Mode_2 aims at detecting faults related to the second type of behaviour mentioned in Sec. 3.1. This test mode detects the same errors detected with the Test Mode_1 plus the faults associated to unexpected messages. The *SelectTM2TxMsg* function is used, and it replaces the *SortNextTxMsg* used in the random algorithm.

```
σ ∈ traces(C)
SendAReplyMsg ∈ W_T
SendAReplyMsg = TRUE /* controlling flag that switches between actions of A_ψ(g) and A_φ(g) */

SelectTM2TxMsg( g ∈ G_C )
{
.       σ = { }
.       ClassifyActions( g )                    /* build sets Ψ, Φ and Γ using actions of O_E on the state g */
.       if A_φ(g) ≠ ∅ and ( not SendAReplyMsg or A_ψ(g) = ∅ )
.       .   .   a ∈ A_φ(g)                        /* random selection of action a from A_φ(g) */
.       .   else if A_ψ(g) ≠ ∅ and SendAReplyMsg and A_φ(g) = ∅
.       .   .   .   a ∈ A_ψ(g)                    /* random selection of action a from A_ψ(g) */
.       .   .   else if A_γ(g) ≠ ∅ and A_ψ(g) = ∅ and A_φ(g) = ∅
.       .   .   .   .   a ∈ A_ψ(g)                /* random selection of action a from A_γ(g) */
.       .   .   .   else return FALSE
.       SendAReplyMsg = not SendAReplyMsg              /* switch the flag state */
.       FindAction( g, a, O_E )                 /* define the trace σ from state g to the action a of O_E */
.       return TRUE
}
```

In order to enable the implementation evaluation, tests have to make the implementation behaviour observable. The *SelectTM2TxMsg* does this task by alternating the selection of A_ψ and A_ϕ actions, when they exist. For that purpose, the variable $SendAReplyMsg \in W_{\mathbf{T}}$ is used, and it controls the selection criteria.

3.4 Test Mode_3

The Test Mode_3 aims at detecting faults related to the third type of behaviour mentioned in Sec. 3.1. This test mode enables the evaluation of implementation behaviours when they are submitted to invalid or unexected messages. The *SortNextTxMsg* in the random algorithm is replaced in this test mode by the function *SelectTM3TxMsg*.

```
σ ∈ traces(C)
SendAReplyMsg ∈ W_T
SendAReplyMsg = FALSE /* controlling flag that switches between actions of A_ψ(g) and A_γ(g) */

SelectTM3TxMsg( g ∈ G_C )
{
.       σ = { }
.       ClassifyActions( g )                /* build sets Ψ, Φ and Γ using actions of O_E on the state g */
.       if A_ψ(g) ≠ ∅ and ( SendAReplyMsg or A_γ(g) = ∅ )
.       .   .   a ∈ A_ψ(g)                    /* random selection of action a from A_ψ(g) */
.       .   else if A_φ(g) ≠ ∅ and A_γ(g) ≠ ∅ and A_ψ(g) = ∅
.       .   .   .   a ∈ A_φ(g)                /* random selection of action a from A_φ(g) */
.       .   .   else if A_γ(g) ≠ ∅ and ( A_ψ(g) = ∅ or ¬SendAReplyMsg )
.       .   .   .   .   a ∈ A_γ(g)            /* random selection of action a from A_γ(g) */
.       .   .   .   if A_γ(g) = ∅ and A_ψ(g) = ∅ and A_φ(g) = ∅
.       .   .   .   .   return FALSE
.       SendAReplyMsg = not SendAReplyMsg          /* switch the flag state */
.       FindAction( g, a, O_E )     /* define the trace σ from state g to the action a of O_E */
.       return TRUE
}
```

The *SelectTM3TxMsg* behaviour is similar to the previous one, since the variable $SendAReplyMsg \in W_{\mathbf{T}}$ also controls the selection of A_γ and A_ψ actions, when they exist.

4 PROFYT

The test methodology presented in this paper lead to the development of a test tool based on the SPIN [6] verifier: the PRogressive On-the-FlY Tester (PROFYT). This tool requires a closed specification model described in the Promela language [6]. In order to close the model, the environment processes must be specified. Based on this model an executable tester is built that operates using the algorithms described above. The tester also includes driver and interface capabilities, which enable the interoperation with the *iut*.

5 Evaluation and Results

In order to evaluate our methodology, we tested some Conference Protocol [11] implementations. The conference protocol entities (CPEs) are the entities responsible for providing the conference service.

5.1 CPE Implementation Under Test

The conference protocol entities (CPEs) under test were implemented in the C programming language. The CPE has two interfaces: the CPE Service Access Point (CSAP) enabling the communication between an user and the CPE processes, and the UDP Service Access Point (USAP) enabling the communication between the CPE processes and the UDP service layer. Different CPEs were tested; each one is a mutant constructed by adding errors to a correct implementation. The conference protocol distribution provides multiple implementation mutants containing faults, from which a total of 29 mutants were tested with PROFYT. For each one, 200 tests with different seeds were executed.

5.2 CPE testing

Testing with the PROFYT tool demands the use of the four test modes (Test_Mode_1, Test_Mode_2 and Test_Mode_3 and random). According to the test methodology presented, the test of each mutant starts with the application of the Test_Mode_1. This type of test enabled the detection of faults in 22 mutant CPEs. The Test_Mode_2 is initiated when the operator assumes that most of the faults detectable with the Test_Mode_1 were detected and removed. The Test_Mode_2 enabled the detection of 4 mutants. The Test_Mode_3 enabled the detection of the remaining 3 mutants.

In order to characterise the value of test modes 1 to 3, the random test mode was also executed and applied to every mutant. The mutants are identified by the numbers defined in the conference protocol distribution. In order to compare de test modes, each mutant was tested 200 times for each test mode, using 200 different seeds. The comparison of the test modes is made based on the number of messages exchanged between the *iut* and the tester before the fault is detected; the number of message exchanged are given as mean and standard deviation values.

The results obtained enable us to conclude that test modes 1, 2, and 3 exhibit shorter traces than the pure random test mode. Instead of a pure the random test mode, these modes enable the reduction of the number of messages exchanged between the tester and iut in about 60% and a reduction of about 70% in the corresponding standard deviation. These values enable us to conclude that, by using the test modes 1 to 3, we can keep some of the random nature of testing and, at the same time, we introduce also some control on the test execution, which enables a significant reduction on the number of messages required to find a fault.

Table 1. PROFYT vs TorX

	PROFYT/TorX			
	$Avr(\frac{Avr}{Avr})$	$Average$	$Avr(\frac{Sdv}{Sdv})$	$Std.Dev.$
Test_Mode_1	69%		44%	
Test_Mode_2	54%	**65%**	44%	**45%**
Test_Mode_3	60%		47%	

5.3 PROFYT vs TorX testing

In order to evaluate the PROFYT performance, we compared it with a similar tool. The TorX tool [11] was chosen for this comparison. Since both PROFYT and TorX use random test approaches, the comparison of their performance was made based on multiple test runs. For each mutant, 200 tests with different seeds were executed. The mean number of messages exchanged between the tester and the implementation were considered as comparison metric.

Table 1 summarises the test results by comparing the average number of messages exchange with the mutants and their standard deviations, on a test mode basis. It also provides the mean ratios on averages and standard deviations, by test modes. The average value represents the relation between the mean values of test sequence lengths obtained with the PROFYT and TorX tools, and expresses a reduction of 35% (100%-65%) on test lengths by testing with PROFYT. The standard deviation of the message sequences length is reduced in 55% (100%-45%) by using the PROFYT.

6 Conclusion

This paper addresses the problem of testing implementations on-the-fly using random model searches. This test approach is sometimes referred as uncontrolled, since there is no human interference on its execution. Although this test can exercise the complete reachability graph, it has limitations such as the large number of messages exchanged for detecting the faults and the difficulty in reproducing the faults.

In this paper, we presented a test method that minimises these drawbacks, while maintaining the essential of random model exploration. The method defines 3 modes which enable to focus the testing in 3 type of behaviours commonly observed by the operators. Although random, the selection of tester messages is constrained by the test type. In this way, the tester messages that are more relevant for each test type are selected first, minimising the number of messages exchanged.

The *iut* conformance testing process is carried out progressively, by executing all the test modes. The progressive approach enables the addressing of fault domains but, simultaneously, it avoid to explicitate individual test purposes. The method is particularly interesting for development phases, where it enables an incremental confidence on the implementation. Besides, by keeping the random component on the algorithms, it enables to enlarge test coverage.

References

[1] P. Ammann, P. E. Black, and W. Ding. Model Checkers in Software Testing. NIST-IR 6777, February 2002. NIST.

[2] E. Brinksma, L. Heerink, and J. Tretmans. Developments in Testing Transition Systems. In M. Kim, S. Kang, and K. Hong, editors, *Tenth Int. Workshop on Testing of Communicating Systems*, pages 143–166. Chapman & Hall, 1997.

[3] J. Callahan, F. Schneider, and S. Easterbrook. Automated Software Testing using Model Checking. In *Proceedings 1996 SPIN Workshop*, aug 1996.

[4] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.

[5] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Lecture Notes in Computer Science*, volume 1687, Toulose, France, September 1999. ESEC/FSE '99, Springer-Verlag.

[6] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[7] A. Kerbrat and I. Ober. Automated test generation from SDL/UML specifications. In *The 12th International Software Quality Week*, San Jose, California, May 1999.

[8] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[9] A. K. R. Groz, T. Jeon. Automated Test Generation from SDL. In Y. L. e. R. Dssouli, G. vonBochmann, editor, *SDL'99 The Next Millenium*, pages pages 135–152, Montreal, Quebec, June 1999. 9th SDL Forum, Elsevier.

[10] J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[11] R. D. Vries and J. Tretmans. On-the-fly Conformance Testing Using Spin. In *4th SPIN Workshop*, ENST, Paris, France, November 1998.