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 \( 1 \leq k \leq N_m \); \( \# c_m \) 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 \( v_{i_l} \); the state of the \( x \) machine variables is jointly represented by \( w = (v_1, ..., v_k, ..., v_w) \), being \( W \) the set of all possible \( w \), and \( w, w \in W \). \( Q \) 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 \( g_t = (q_t, (c_1, ..., c_m), (v_1, ..., v_k, ..., v_w)) \), where \( g_t \in G \).

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 \( A_P = I_P \cup \mathcal{O}_P \cup \mathcal{W}_P \cup \)
to queues, and output symbols, representing the transmission of messages reception of messages from the queues. $O_P$ is the set of output symbols, representing the transmission of messages to queues, and $I_P \cap O_P = \emptyset$. $W_P$ is the set of symbols denoting the operations over the machine variables. $\tau$ labels the transitions between states with no execution of actions in $I_P$, $O_P$, or $W_P$. The execution of actions in $I_P$ or $O_P$ 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 $g_{i+1}$ to state $g_{i+1}$. This transition is represented by $l((g_{i}, a) = g_{i+1}$, or $(g_{i}, a, g_{i+1}) \in T_P$.

For each action $a \in I_P \cup O_P$, a unique $d(a)$ identifies the queue used by the action $a$; the message transferred through the $a(a)$ 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 $a \in I_P$ is possible or executable when $d(a)$ queue is not full, i.e. $\#d(a) \neq 0$ and $msg(a) = u_{d(a)}(1)$. In state $g_{i+1}$, after the execution of the action $a$, state $c_{d(a)}$ is defined by $c_{d(a)} = c_{d(a)} \cup \{u_{d(a)}\}$. A transition resulting from an action $a \in O_P$ is executable when $d(a)$ queue is empty, i.e. $\#d(a) < N_d(a)$ and $msg(a)$ in $U_{d(a)}$. In this case the new state of queue $d(a)$ is defined by $c_{d(a)} = c_{d(a)} \cup \{u_{d(a)}\}$, where $u_{d(a)} = msg(a)$ and $k + 1$. A quiescent state $g_X$ is a state having only outgoing transitions labelled with input actions. The set of quiescent states of machine $P$ is defined by:

$$\Delta_P = \{ g \in G_P \mid \forall a \in (A_P \setminus T_P) : l((g, a) \not\in G_P) \}$$

$A_P^*$ 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 $a \in A_P$. The length of the $a$ trace is represented by $|a|$. The concatenation of two traces $\sigma_a$ and $\sigma_b$ is represented by $\sigma_a \sigma_b$, while $\sigma_a$ denotes the concatenation of trace $\sigma_a$ with the action $a$. Moreover, a function $tail(\sigma)$ identifies the last action of $\sigma$, such that $tail(\sigma) = a$. The function $head(\sigma)$ identifies the first action of $\sigma$, such that $head(a, a) = a$.

Moving from a state by executing a trace leads to extended transitions $T_P \subseteq G_P \times A_P \times G_P$ by $l((g, a, g') \in T_P)$. The set of all the traces defined in $P$ is represented by $traces(P)$.

The composition of machines $X = (G_X, g_0, A_X, T_X, M_X)$ and $Y = (G_Y, g_0, A_Y, T_Y, M_Y)$ is defined by a machine $Z = (G_Z, g_0, A_Z, T_Z, M_Z)$ where $G_Z = G_X \times G_Y$. (Since $G_Z = Q_X \times G_Y$ and $Q_Z = Q_X \times Q_Y$, $C_Z = C_X \times C_Y$, and $W_Z = W_X \times W_Y$; $g_0$ is the initial state, defined by $g_0 = (g_0, X, g_0, Y)$; $A_Z = A_X \cup A_Y = T_Z \cup O_Z \cup W_Z \cup \tau)$ is the set of actions of $Z$, in which $T_Z = T_X \cup T_Y$, $O_Z = O_X \cup O_Y$, and $T_Z \cap O_Z = \emptyset$. $W_Z = W_X \cup W_Y$ is the set of symbols representing the manipulation of $Z$ variables; $T_Z \subseteq G_X \times A_Z \times G_Y$ 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 $((g_{i+1}, g_Y), a, (g_{i+1}, g_Y), g_{i+1})$ is defined in $T_Z$ if either:

1. $\forall a \in T_X \cup O_X, (g_{i}, a, g_{i+1}) \in T_X, \exists g_Y, g_{i+1} \in G_Y : (d(a) \in MY \land (g_Y, \tau, g_{i+1}) \in T_Y) \lor (d(a) \in MY$ and $g_Y = g_{i+1});$ or

2. $\forall a \in T_Y \cup O_Y, (g_Y, g_{i}, g_{i+1}) \in T_Y, \exists g_Y, g_{i+1} \in G_Y : (d(a) \in MX \land (g_X, \tau, g_{i+1}) \in T_X) \lor (d(a) \in MX \land g_X = g_{i+1});$ or

3. $\forall a \in A_X \cup O_X \cup O_Y \cup g, \sigma \in I \cup O \cup S, T \subseteq G_X \cup G_Y$.

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 $((g_{i+1}, g_Y), a, (g_{i+1}, g_Y), g_{i+1})$ is defined in $T_Z$ by replacing the conditions 1. and 2. mentioned above by 1. $\forall a \in O_X, b \in T_Y, \exists g_Y, g_{i+1} \in T_X, (g_{i+1}, b, g_{i+1}) \in T_Y : d(a) = (b, \emptyset) \land msg(a) = msg(b);$ or 2. $\forall a \in I_X, b \in O_X, \exists g_Y, g_{i+1} \in T_X, (g_{i+1}, b, g_{i+1}) \in T_Y : d(a) = (b, \emptyset) \land msg(a) = msg(b)$.

The set of $X$ states resulting from the composition of quiescent states of $X$ is $\Delta_{X} = \{ g \in G_Z \mid \forall a \in D_X, g \in G_Y : g = (g_X, g_Y) \}$

### 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 $S = (G_S, g_0, A_S, T_S, M_S)$.

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 $E = (G_E, g_0, A_E, T_E, M_E)$. The actions of $A_E$ are either message transmissions or receptions, and are related with the transmissions and receptions of $S$. The set $O_E$ is defined by $O_E = \{ a' \in A_E \land msg(a) = msg(a') \land a' \not\in O_S \}$ and the set $I_E$ of reception actions is given by $I_E = \{ a' \in O_E \land msg(a) = msg(a') \land a' \not\in I_S \}$. The transitions of $E$ satisfy the condition $\forall a \in A_E : (g_0, a, g_0) \in T_E$. The set $M_E$ is given by $M_E = \{ d(a) \mid a \in I_E \cup O_E \}$.

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 $C = (G_C, \delta_0, \Sigma_C, T_C, M_C)$. The behaviour of the tester is inferred from $C$. 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 $C$; the reception of a message by the tester is possible only if the reception of the message is also possible on $C$.

The tester $T$ can be described by $T = (G_T, g_{0T}, A_T, T_P, M_T)$ where $G_T = (Q_T \times C_T \times W_T) \cup \{(\text{pass}, \text{fail})\}$ is the set of $T$ states; $g_{0T} = g_{0C}$ is the initial state of $T$; $A_T = T_P \cup C_T \cup W_T \cup \{\tau\}$ is the actions set; $T_T \subseteq Q_T \times C_T \times G_T$ is the set of $T$ transitions; $M_T$ is the set of $T$ queues. $G_T$ represent the actions of $O_T$. $T_T$ includes also two additional input actions, $I_T = I_E \cup \{\xi, \delta'\}$; $\xi$ represents the reception of an unknown messages on the implementation under test ($iut$). The queues $M_T$ are replicas of the queues $M_E$; however, the vocabulary of $M_T$ queues is larger than the vocabulary of $M_E$ queues, in order to accommodate the invalid $iut$ messages. The $G_T$ and $T_T$ 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 $I = (G_I, g_{0I}, A_I, T_I, M_I)$, and assume that $M_I = M_T$, in order to enable the interoperation between $I$ and $T$.

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

$$
T_T = \emptyset \quad \text{\textsuperscript{1}} \quad \text{\textsuperscript{1}\textsuperscript{1} set of $T$ transitions}
$$

$$
(\text{GetQueuesWithMessages}) \quad \text{\textsuperscript{1}\textsuperscript{1}\textsuperscript{1} get the set of queues having messages from the $iut$}
$$

$$
\text{if } \exists \delta \notin g_{0T} \text{ and } g_{0C} \text{ and } m \in \text{QueuesWithMessages} \quad \text{\textsuperscript{1}\textsuperscript{1}\textsuperscript{1} update the set QueuesWithMessages}
$$

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

The set $\text{QueuesWithMessages}$ is a list of queues containing messages to be consumed by $T$; this set is defined by the function GetQueuesWithMessages(), and it is rebuilt after the reception or transmission of every message.

When the $\text{QueuesWithMessages}$ set is empty, the quiescence of $I$ is verified in $C$, by the function IsQuiescent(), that searches $C$ for an action $r \in I_E$; a quiescence is valid if $r$ is unreachable. If the quiescence observed is not specified, i.e. if there is at least one reachable $r \in I_E$, a new transition leading the test to the $\text{fail}$ state is added to $T_T$. The action associated to this transition is $\delta'$, denoting the invalid quiescence detection, and terminating the test. When the detected quiescence is valid, the test $T$ selects a message to transmit to $I$. The selection of this message is performed by the function SortNextTxMsg(), which searches the model $C$ for an action in $O_E$. The trace $\sigma$ leading to the selected action is used to guide the $C$ execution. The extended transition, from state $g_{0T}$, and the actions of $\sigma$, is computed in order to refresh the state of $C$. Based on the action selected, a new state is added to $G_T$, and a transition to this state is added to $T_T$. 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 $\text{QueuesWithMessages}$ is rebuilt.

The function RcvdMsgIsValid() is applied to the first message of queue $m \in \text{QueuesWithMessages}$, which searches for a $\sigma$ trace in $C$, guiding the machine toward an action of $I_E$ 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 $G_T$, the action of $I_T$ labelling the reception of that message is selected and, using them, a new transition is created in $T$ that accepts the message and leads the tester to a new state.

The test continues with the execution of $\sigma$ actions on the $C$ machine and, by executing the $RcvMsg(\ m\ )$ function, removes the message from queue the $m \in M_T$. Then, the $QueuesWithMessages$ set is rebuilt, and the next message in queue is evaluated. If the $\sigma$ 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 $T$ leads the machine $C$ to a state $g \in \Delta^2_C$, and also $t$ output actions in $O_E$ matching all the implementation inputs specified for a state $g$. Each action can initiate three classes of traces:

- **i** $\Psi$ traces lead $C$ to the input actions $r \in I_E$; the set $\Psi(g, t)$ contains the $\Psi$ traces initiated with the action $t$ on state $g$. The $t$ actions initiating the $\Psi$ traces belong to the set $A_{\psi}$, defined by $A_{\psi}(g) = \{ t \in O_E | \exists g \in \Delta^2_C : \Psi(g, t) \neq \emptyset \}$.
- **ii** $\Phi$ traces lead $C$ to the quiescent states $g_\epsilon \in \Delta^2_C$; the set $\Phi(g, t)$ contains the $\Phi$ traces initiated with the action $t$ on state $g$. The actions $t \in O_E$ initiating the $\Phi$ traces belong to the set $A_\phi$, defined by $A_\phi(g) = \{ t \in O_E | \exists g \in \Delta^2_C : \Phi(g, t) \neq \emptyset \}$.
- **iii** $\Gamma$ traces have length 1 and do not change the state of $C$, or lead $C$ to quiescent states, for which no judgement is possible. The $\Gamma(g, t)$ set contains the $\Gamma$ 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 $O_E$ starting the traces of $\Gamma(g, t)$ belong to the set $A_\gamma(g)$ defined by $A_\gamma(g) = \{ t \in O_E | \exists g \in \Delta^2_C : \Gamma(g, t) \neq \emptyset \}$.

### 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 $SelectTM1TxBMsg$, instead of the $SortNextTxBMsg$, presented above.

$$\sigma \in traces(C)$$

```plaintext
select:\ SelectTM1TxBMsg( \sigma \in G_E )
{
    \sigma = \{ \}
    - ClassifyAction( \sigma )
    - if $A_{\phi}(g) \neq \emptyset \quad \text{\sigma random selection of action from } A_{\phi}(g)$
    - else if $A_{\gamma}(g) \neq \emptyset \quad \text{\sigma random selection of action from } A_{\gamma}(g)$
    - else return fail
    - FindAction( g, a, O_C ) /* define the trace $\sigma$ from state $g$ to the action $a$ of $O_C$ */
    return $\sigma$
}
```

The $SelectTM1TxBMsg$ function starts with the classification of the test actions executable from state $g$, and their distribution by the sets $A_{\phi}$, $A_\gamma$, and $A_{\phi}$, 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 $\sigma$ trace, that will be used to drive the machine $C$ 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 $SelectTM2TxBMsg$ function is used, and it replaces the $SortNextTxBMsg$ used in the random algorithm.
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 $A_\phi$ and $A_\psi$ actions, when they exist. For that purpose, the variable $SendAReplyMsg \in W_T$ 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 unexpected messages. The SortNextTxMsg in the random algorithm is replaced in this test mode by the function SelectTM3TxMsg.

\[
\sigma \in \text{traces(C)} \\
\text{SendAReplyMsg} \in W_T \\
\text{SendAReplyMsg} = \sigma \uparrow \text{controlling flag that switches between actions of } A_\phi (g) \text{ and } A_\psi (g) \\
\text{SelectTM3TxMsg}(g \in G_C) \\
\{ \\
\sigma = () \\
\text{ClassifyActions}(g) \\
\text{if } A_\phi (g) \neq \emptyset \text{ and } (\text{SendAReplyMsg} \lor A_\phi (g) = \emptyset) \\
\text{and } A_\psi (g) \neq \emptyset \text{ and } (\text{SendAReplyMsg} \lor A_\psi (g) = \emptyset) \\
\text{else if } A_\phi (g) \neq \emptyset \text{ and } A_\psi (g) = \emptyset \\
\text{else if } A_\psi (g) \neq \emptyset \text{ and } A_\phi (g) = \emptyset \\
\text{else if } A_\phi (g) = \emptyset 	ext{ and } A_\psi (g) = \emptyset \\
\text{else if } A_\phi (g) = \emptyset 	ext{ and } A_\psi (g) = \emptyset \\
\text{return false} \\
\text{SendAReplyMsg} = \text{not SendAReplyMsg} \\
\text{σ} \text{ switch the flag state} \\
\text{PinAction}(g, A_\phi) \text{ define the trace } σ \text{ from state } g \text{ to the action } A_\phi (g) \\
\text{return TRUE} \\
\}
\]

The SelectTM3TxMsg behaviour is similar to the previous one, since the variable $SendAReplyMsg \in W_T$ also controls the selection of $A_\phi$ and $A_\psi$ 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-FIY 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.
5.3 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 explicite 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


