8

DECISION ON TESTER CONFIGURATION FOR MULTIPARTY TESTING

Maria Törö
Computer Science Department
The University of British Columbia
2366 Main Mall, Vancouver, B.C., V6T 1Z4 Canada
toeroe@cs.ubc.ca

Abstract Communication systems are composite systems which can conduct simultaneous communications with their environment. Such systems may require multiparty testing to determine their conformity.

To develop an appropriate test suite, one should establish the required test method first. To decide whether multiparty testing is necessary, that is, whether the system truly accepts simultaneous signals from the environment, we describe the interconnection of system interfaces with internal buffers by a matrix, assuming the system structure and the way of communication of SDL. From this matrix we try to select independent signals, which can be accepted simultaneously by the system. The criterion of signal independence is that signals should be offered to different buffers. From sets of independent signals we generate composite inputs. We suggest to use these composite signals as selection criteria to develop the set of test purposes and apply them to generate a test suite. Accordingly we suggest a modification to the strong reasonable environment technique, most often used to generate test suites. We demonstrate the method on the foreign agent part of the IP Mobility Support.

Keywords: SDL, communicating finite-state machines, interface-buffer interconnection matrix, required test method, reduced reachability tree

1. INTRODUCTION

Real protocol implementations are typically composite systems and can communicate simultaneously with several entities. However, most of the theoretical works done on conformance testing, were restricted to a single or, if multiple entities were involved, to a sequential communication with those entities. Only such sequential test cases could be specified in the first version

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35567-2_25

G. Csopaki et al. (eds.), *Testing of Communicating Systems*

of TTCN [CTMF], where all the communication between the tester and the system under test (SUT) was described as a single sequence, even if it involved multiple entities connected through different PCOs (point of control and observation). This lack of concurrency on the tester side resulted in the introduction of the methodology of multiparty testing. Accordingly TTCN was extended to concurrent TTCN [TTCNe] that would allow to specify multiple test components that makes the tester capable to send and receive simultaneous events. multiparty testing is still an open research area.

In all cases, when a specification defines several interfaces between a system and its environment one must consider multiparty testing, because an implementation of such a system might be able to accept and to process concurrent signals. Only multiparty testing can provide simultaneous inputs toward the SUT, although it can still not guarantee that these events will occur indeed concurrently.

On the other hand, having several interfaces does not mean in all cases that signals provided simultaneously through them are indeed processed concurrently by the system. To decide whether multiparty testing is really necessary, one needs to take a closer look at the internal structure of the system specified, and the way it communicates to its environment. This greatly depends on the underlaying model of the formal description technique used for specification, provided that implementations follow both the specified structure and features of the model.

A system internally can be specified as a single-entity system; or it can be distributed, and composed of multiple entities working in parallel and communicating with each other and possibly with the environment.

A component's behavior can be sequential, i.e. it accepts only one stimulus at a time; or it can synchronize on its inputs, i.e. waits until all required stimuli have been received. An example of the first case is the (extended) finite-state machine model, which defines the behavior as a reaction on a single input received in a given state. An example of the second case is the Petri-net model. In Petri-nets a transition may have several input places, and the transition may fire only when each input place holds at least one token.

The communication between the entities of a system can be synchronous, such as in LOTOS, where participating processes are blocked until the required synchronization occurs; or asynchronous as it is in SDL [SDL], where signals are buffered in queues until consumption.

All these and the specified system itself determine whether multiparty testing is a must. These factors define the required test method, which in turn, guides the test suite generation.

The goal of our paper is to take a closer look at SDL and systems specified in SDL to decide whether multiparty testing is inevitable. We present a matrix

which helps us to decide what test method and tester configuration should be applied to a system. Then the result is used for selection of test purposes and in the generation of test cases. The novelty of our approach is that it takes a structural approach. It investigates first the structure of the system specified, then uses the results in the subsequent steps of test suite generation. Until now the question of determining of the required test method was hardly ever discussed further than enlisting the abstract test methods of appropriate standards [CTMF, FMCTa].

According to SDL, we assume a compound system of asynchronously communicating finite state automata. To determine the required test method, we describe the interconnection of the system interfaces and the internal buffers of the system by a matrix. From this matrix we derive the independent signal sets, from which we generate composite inputs used to control multiparty testing. The main criterion of signal independence is that signals are offered to different buffers. According to the methodology of multiparty testing, for each signal in the set we need to provide a separate test component in the tester. Also, the test generation method should take into account these simultaneous events. Hence we suggest to build a reduced reachability tree by applying a composite input (i.e. a set of simultaneous signals at the selected interfaces) at a time, whenever the system is in a stable global state.

Our paper presents this approach as follows: First, we give an overview of test generation methods used for systems specified in SDL. We conclude this overview with our approach to the test suite generation. In the third section, we introduce a system of communicating finite-state machines (CFSM) as the basis of our further discussions. In section four, we analyze the features of SDL to determine the required test method for a system specified in SDL. We introduce the interface-buffer interconnection matrix, from which we derive sets of composite inputs are for multiparty testing. In section five a modification of the algorithm of the generation of the reduced reachability tree is suggested. Finally, section six presents an example: the model and the reduced reachability tree for the foreign agent part of the IP Mobility Support protocol.

2. TEST GENERATION METHODS FOR COMPOSITE SYSTEMS SPECIFIED IN SDL

For standardized FDTs draft standards [FMCTa, FMCTb] suggest several test generation methods. Here we discuss only methods suggested for SDL.

Most of the methods suggested for SDL, are based on a predefined set of test purposes given in MSC (Message Sequence Chart) [FMCTb, Ek97]. These methods generate the reachability tree of a system specified, then they

compare it with the set of test purposes. Whenever a part of the reachability tree covers a given test purpose scenario a test case is generated. The pre- and postambles, and the alternative behaviors are derived from the reachability tree. The *pass* verdict is assigned to the branch covering the test purpose, *inconclusive* verdicts are assigned to other behaviors derivable from the reachability tree, and a *fail* verdict is assigned to everything else, i.e. to behaviors which cannot be derived from the specification. According to the literature and tool descriptions, implementations of this method take into consideration mostly single-process systems, but never more than a single input queue between the system and its environment (although this input queue may be connected to several interfaces) [FMCTb, Fern96b, Grab97]. In this methodology the set of test purposes is derived from the requirements specified for the protocol in the standard.

Given a (standard) protocol specification in SDL, it is already a requirement specification that an implementation should satisfy. Indeed, it is the most detailed set of requirements that ever would be imposed by a standard. Any set of test purposes will only be a subset of this specification if it does not cover the full specification (in which case there is no reduction of complexity). The question, whether a subset of requirements is sufficient to determine conformity, rises all the time. In addition, there is no established methodology to derive the test purposes from a system specification and it is mostly done by experts:

A protocol/system lifecycle starts with the specification of the requirements, for example, in MSC. These message sequence charts are then used (1) to produce a skeleton of the system specification in SDL which will be refined then until the necessary level of details is achieved. The requirements also (2) compose the basis for test purpose definitions. Unfortunately this branching in the lifecycle means that the original set of requirements and the specification might not be consistent any more. The refinement of the specification introduces new parts to the system behavior which might not be covered by the original requirements. That is, the original set of requirements cannot be used directly as set of test purposes. They have to be re-generated from the SDL specification, or at least validated and verified against the final system specification. Hence the question is: how to select the appropriate (sub)set of requirements from the specification?

[Luo94a, Tan96] describe a different approach of test suite generation for SDL: A transformation of the SDL system into a system of communicating finite-state machines. Unfortunately the transformation of the SDL system cannot be generalized. Not all SDL systems can be transformed into an equivalent system of communicating finite-state machines, even when only the control part of protocols is considered.

[Fern96a, Fern96b, Grab96] apply verification techniques to test suite generation.

The method presented in [Fern96a, Fern96b] again uses test purpose specifications to generate the test suite. The algorithm interleaves the events at different PCOs, which is not necessary when one is allowed to define parallel test components. The goal of the introduction of concurrent TTCN and parallel test components was to avoid this complexity. The method, as presented, deals primarily with the control part of a specification though it does not exploit this assumption.

[Grab96] discusses different methods used for verification of SDL specifications to reduce the complexity of the state space exploration, and their application to test generation methods. In SDL the buffer size is unlimited and may be infinite. This feature makes the state space infinite, therefore a reasonable reduction technique is required. [Grab96] concludes that the most effective reduction technique for SDL is the strong reasonable environment, i.e. the application of a single external stimulus and only at a stable global state of a system. Similarly as it is used in the random walk methods (guided [Lee96], weighted [Kang97]). Obviously, this technique excludes simultaneous inputs what is necessary for multiparty testing. Therefore we suggest to modify this technique to adjust it to multiparty testing:

That is, we allow at most one external signal at a time *for each point of control*. This means that if there are several PCOs, instead of a single external signal, a set of external signals is applied to the system at a time, but no more than one signal at each PCO. We apply a single set of input signals only to a stable global state of the system, i.e. all input queues are empty, so does the original method. Note that this requirement puts a limitation on the delay introduced by delaying channels of SDL, since a stable global state is reached only when all these buffers are emptied. The question is what sets of external inputs should be used to build the reachability tree. Is there any possibility to reduce this set compared to the set of all possible combinations of inputs?

In addition random walk methods avoid the state space explosion by not creating the composite automaton for a CFSM system. They make the assumption, that stable global states of the implementation are observable. This may not be the case at the test campaign, however, one can safely assume the same at the test suite generation without introducing any limitation regarding the implementation. At test suite generation, one can observe internal queues and states of the specification and determine the stable global states. Since these events are invisible and uncontrollable for a tester at the test campaign, one may only observe and should not control the internal communication even at the test suite generation. Thus we have to evaluate all possible ordering of simultaneous signals in internal input queues.

Note: It is not our goal to introduce any new test suite generation technique. We aim merely at answering the question whether multiparty testing is necessary and if it is, what abstract test method should be used. We suggest to use the result to select test purposes and to produce a test suite by existing methods. We only demonstrate our approach on the method producing a reduced reachability tree and specifying a test case for each branch of the tree. For our demonstration we use a model of communicating finite-state machines as follows.

3. COMMUNICATING FINITE-STATE MACHINES WITH EXTERNAL INTERFACES

We simplify the SDL model to a system of n communicating finite-state machines, which has m external interfaces: $CFSM = (\{M_1, M_2, ..., M_n\}, \{I_1, I_2, ..., I_m\})$. An automaton can have a dedicated interface or multiple interfaces or share an interface with other automata.

Automata communicate asynchronously, thus each automaton has a single input queue, which collects all inputs signals for that given automaton. An automaton accepts input signals from automata of the system and from the system environment.

An automaton of the system is described as $M = \langle S, E, O, tr, s_0, q \rangle$, where

S - the finite set of states of the automaton;

E - the finite set of input signals of the automaton. It is composed of two sets $E = L \cup E^i$ where L is the set of external input signals and E^i is the set of internal input signals;

O - the finite set of output signals of the automaton; similarly to input signals, it is composed of two sets $O = O^e \cup O^i$ where O^e is the set of external output signals and O^i is the set of internal output signals;

tr - the transition function of the automaton;

s_0 - the initial state of the automaton, $s_0 \in S$;

q - the input queue of the automaton.

The transition function is given as $tr = (s_s, \iota, \{q_i;\omega_i ..., q_x;\omega_x\}, s_e)$, and it determines for the (s_s, ι) state-input pair $s_s \in S$, $\iota \in E$ the resulting outputs $\omega_i, ..., \omega_x \in O$ appended to appropriate queues $(q;\omega)$, and the next state $s_e \in S$ of the automaton. An input from the environment is put directly to the input queue of the appropriate automaton. To model delaying channels of SDL we allow spontaneous transitions, i.e. no input signal is required to trigger the

transition, which leads to a new state ($tr_\varepsilon = (s_s, \varepsilon, \{q_j;\omega_j \ldots, q_x;\omega_x\}, s_e)$, where $s_s \neq s_e$). An automaton modeling a delaying channel is shown in Figure 1a. It has two states: s_0 - delivering state (initial) - and s_1 - blocking state. The transitions between them are spontaneous transitions. In the blocking state the automaton introduces a delay; in the delivering state it forwards signals. A timer can be modelled in a similar way (see Figure 1b). It is an automaton with two states: s_0 - timer does not run - and s_1 - timer runs. From s_0 state the timer is set by the set signal and moves to the s_1 state, from which a spontaneous transition or the reset signal leads back to the s_0 state. In s_1 state the set signal triggers a null transition.

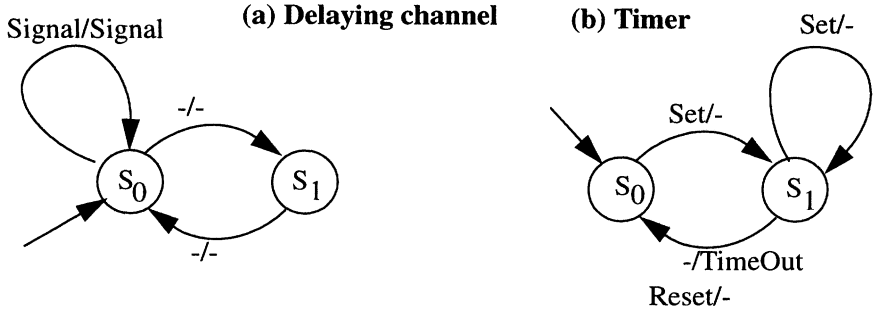


Figure 1 The timer and the channel automata

An interface of the system is composed of two parts, one for each direction $I = (Inp, Out)$. The input part $Inp = \{(q_i, L_i) | 1 \leq i \leq n, L_i \subseteq E_i\}$ is the list of queues to which the interface delivers signals and the set of carried signals for each queue. The output part $Out = (q, O^e)$ is the output queue and the set of output signals at this interface $O^e \subseteq \bigcup_{i=1}^{n} O_i^e$.

Such a system may accept simultaneous stimuli. In the next section we discuss whether the application of concurrent inputs, i.e. multiparty testing is necessary.

4. DETERMINING THE REQUIRED TEST METHOD

The required test method depends on the structure and the method of communication of the system to be tested. The abstract test method determines the required test suite generation method. Since we focus on SDL we consider of the structure and the way of communication of such systems. We also

take into account assumptions made by TTCN about the tester configuration:

According to concurrent TTCN for each simultaneous thread of communication with the system under test, one needs a dedicated parallel test component, because each parallel test component is able to control only a single thread of events (though these events may occur at different PCOs). Thus we have to determine the possible concurrent events in the communication with the system.

We assume that a test campaign can and should use only the channels specified between the SDL system and its environment as PCOs. We refer to these channels as *interfaces*. In general a tester offers signals only at interfaces and it observes signals received only at interfaces. It cannot influence or even observe directly any internal communication. Therefore we focus on signals offered at interfaces. They also play an essential role in the test suite generation: They determine the tester configuration required to *control* the test campaign as each simultaneous signal requires a separate test component. The configuration required to *observe* test events is derived from the expected output events, after a test generation method has been applied (e.g. reachability tree) to the specification.

All these suggest that to cover all the possible behavior, one should offer all the combinations of valid input signals to the system at its interfaces. However after a closer look at the system structure it is possible to reduce this estimation:

An SDL process has a single input queue and consumes a single signal at a time from its input queue. All signal routes leading to the process deliver signals to this single input queue, where simultaneous signals are ordered arbitrarily and buffered until consumption. The behavior of the process is sequential. Hence a single test component is sufficient to control a single process if its input queue is exposed directly to the tester. The number of interfaces leading to this process, that is to its input queue, has no effect on this. The question is whether an input queue is exposed directly to the tester.

A process, if applicable, is connected to the system environment by at least a channel and a signal route. Signal routes and non-delaying channels, deliver signals immediately. Therefore one may assume that they deliver simultaneous signals concurrently, i.e. they preserve the concurrency of interfaces. This also means that if a process connected to the environment only by non-delaying channels, it exposes its input queue directly to the environment. Thus the process itself serializes the incoming signals via buffering them in its input queue.

In contrast, a delaying channel acts like an input queue itself. A delaying channel arbitrarily orders simultaneous signals at reception, then it delivers them one by one, after a non-deterministic delay. As we suggested in the pre-

vious section, delaying channels can be represented as predefined automata with a single input queue, thus a single test component is sufficient to control a delaying channel and all the processes connected via it.

That is, we need to investigate the interconnection of interfaces and buffers (input queues of processes and delaying channels) to determine what combination of events may occur concurrently at the interfaces.

Figure 2 shows the three basic cases of interface-buffer interconnections and a combination of them:

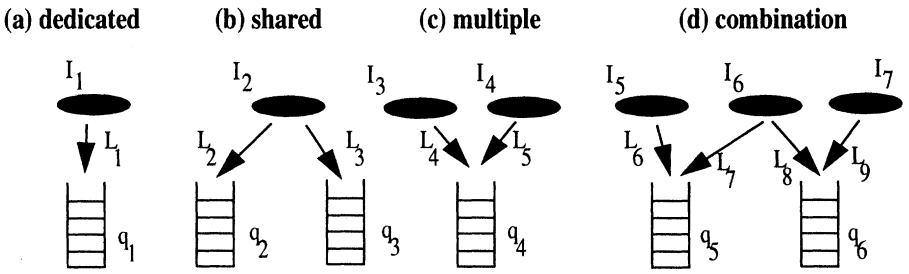


Figure 2 Interface-buffer interconnections

(a) *Dedicated interface*: buffer q_1 have a dedicated interface I_1.

(b) *Shared interface*: buffers q_2 and q_3 share a common interface I_2. We may or may not assume that a single interface can deliver simultaneous signals.

(c) *Multiple interfaces*: buffer q_4 has two interfaces I_3 and I_4. Although the two interfaces can deliver signals simultaneously, the signals will be ordered by the buffer, that is the interfaces exclude each other, i.e. they act sequentially.

(d) *Combination*: To buffers q_5 and q_6 three interfaces deliver signals: I_5, I_6 and I_7. I_5 delivers exclusively to q_5, but I_6 is shared between q_5 and q_6. I_7 is dedicated to q_6. This means that if I_6 delivers a signal either to q_5 or to q_6, only one of the other interfaces can act in parallel: I_7 in the first case, I_5 otherwise. If I_6 is not used I_5 and I_7 can deliver independently from each other. Whether I_6 can deliver to both q_5 and q_6, and exclude the other interfaces is a further issue

We describe the interface-buffer interconnection (R) of a system by a matrix columns of which mean interfaces and rows of which represent buffers. If an interface delivers signals to a given buffer, we put down the appropriate signal list (L) in the cell. Otherwise we put down the $\varnothing$ symbol for the

empty set. For the example of Figure 2(d) the matrix is :

$$R = \begin{array}{c} \begin{matrix} \textbf{interfaces:} \\ \mathbf{I_5} \quad \mathbf{I_6} \quad \mathbf{I_7} \end{matrix} \\ \begin{bmatrix} L_6 & L_7 & \varnothing \\ \varnothing & L_8 & L_9 \end{bmatrix} \end{array} \begin{matrix} \textbf{queues:} \\ \mathbf{q_5} \\ \mathbf{q_6} \end{matrix}$$

To select independent signals, one needs to select independent elements of the matrix.

(1) Since input queues serialize signals, elements of the same row are dependent. To select independent sets one should chose only one signal at a time for each buffer. This means that at most one element per row can be selected.

(2) If we also make the assumption that each interface delivers only one signal at a time, that will further restrict our selection of matrix elements. That is, in addition to the previous restriction of one element per row, one is allowed to select at most one element per column.

These selections result in matrices, rows and columns of which linearly independent from each other.

From matrix R we derived the following matrices:

$$R_1 = \begin{bmatrix} L_6 & \varnothing & \varnothing \\ \varnothing & L_8 & \varnothing \end{bmatrix}, \; R_2 = \begin{bmatrix} L_6 & \varnothing & \varnothing \\ \varnothing & \varnothing & L_9 \end{bmatrix}, \; R_3 = \begin{bmatrix} \varnothing & L_7 & \varnothing \\ \varnothing & \varnothing & L_9 \end{bmatrix}, \; R_4 = \begin{bmatrix} L_6 & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix},$$

$$R_5 = \begin{bmatrix} \varnothing & L_7 & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix}, \; R_6 = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & L_8 & \varnothing \end{bmatrix}, \; R_7 = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & L_9 \end{bmatrix}, \; R_8 = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix}$$

Each of the derived matrices represent a tester configuration for control, since the simultaneous signals can be offered only by parallel test components.

In our example R_1, R_2 and R_3 represent configurations of multiparty testing. Each of them requires simultaneous use of two interfaces: I_5 and I_6, I_5 and I_7, and I_6 and I_7 respectively. Configurations of R_4, R_5, R_6, and R_7 offer a single external signal to the system. Finally in R_8 no interface is selected for control, that allows to observe a behavior triggered by internal events, such as keep-a-live signals, etc.

If we allow multiple signals at an interface, that is we do not make our second assumption about the interfaces, an additional matrix is derived:

$$R_9 = \begin{bmatrix} \varnothing & L_7 & \varnothing \\ \varnothing & L_8 & \varnothing \end{bmatrix}.$$

We construct composite inputs from the possible independent inputs by

selecting one signal from each selected independent signal list. Thus we create the Cartesian product of the selected signal lists of the matrices and create the union of these sets. The set of composite inputs for our example is: $\Lambda = (L_6 \times L_8) \cup (L_6 \times L_9) \cup (L_7 \times L_9) \cup L_6 \cup L_7 \cup L_8 \cup L_9 \cup \varnothing$.

For R_9 this set is extended with the set of $(L_7 \times L_8)$.

5. GENERATION OF THE REDUCED REACHABILITY TREE

As we suggested, we relaxed the strong reasonable environment technique to generate a reduced reachability tree. That is, we build the reduced reachability tree by offering a composite input at a time to the system in a stable global state. We consider a system state as a tuple of the component automata's states coupled with their input queues and output queues of the system:

$\sigma = \langle \langle s_1, q_1 \rangle, \langle s_2, q_2 \rangle, \ldots, \langle s_n, q_n \rangle \rangle$, where S_i is the state, q_i is the input queue of the ith automaton.

A state is a stable global state when all of the input queues of automata are empty, hence such a system state is described as an tuple of the automata's states, i.e. $\sigma^{st} = \langle s_1, s_2, \ldots, s_n \rangle$.

States having spontaneous transitions are stable if the automaton's input queue is empty. However at the generation of the reachability tree one has to treat spontaneous inputs as well. We calculate the successor states for the spontaneous transition i.e. we add to the set of reachable states, states reached by firing the spontaneous transition.

We generate the reduced reachability tree by the following algorithm:

begin

$\Sigma = \varnothing$; // reached stable global states

$\sigma_0^{st} = \langle s0_1, s0_2, \ldots, s0_n \rangle$; // the initial stable global state

$d = 0$; // the current depth of the reachability tree

$\Sigma_0 = \{\sigma_0^{st}\}$; // stable global states at depth 0

do{

$\Sigma = \Sigma \cup \Sigma_d$; // add newly reached stable global states

$\Sigma_{d+1} = \varnothing$;

for ($i = 1, i \leq |\Sigma_d|, i = i + 1$) // for all stable global states of depth d

for ($j = 1, j \leq |\Lambda|, j = j + 1$) { // for all composite inputs

// apply the composite input to the stable global state

// and generate the reachable stable global states:

$$\sigma_i^{st} \xrightarrow{l_j} \{\sigma^{st} | \sigma^{st} = \langle \langle s_1, \varnothing \rangle, \langle s_2, \varnothing \rangle, \ldots, \langle s_n, \varnothing \rangle \rangle\}$$

where $l_j \in \Lambda$, $\sigma_i^{st} \in \Sigma_d$

// store a stable global states iff it has not been reached before:

if $(\sigma^{st} \notin \Sigma)$ then $\Sigma_{d+1} = \Sigma_{d+1} \cup \{\sigma^{st}\}$;

}

$d = d + 1$;

} *while* $(\Sigma_d \neq \varnothing)$; // until new stable global states have been reached

end

To generate all reachable stable global states from a stable global state $\sigma_i^{st} \in \Sigma_d$ by applying a composite input $l_j \in \Lambda$, we do a breadth first exploration (similarly as it is presented in [ObjG]). For each automaton we interleave simultaneous signals from different senders and take into account possible spontaneous transitions. We also interleave signals at the same output queue. According to the algorithm we check each newly reached stable global state with the set of stable global states reached before and if it is already in the set, we truncate the reachability tree.

Note that until now we discussed the tester configuration only with respect to composite inputs, i.e. from the point of view of control. We determine the configuration for observation from the generated output events each time we reach a stable global state: We define an observer at each output queue which contains output signals. Any time a control point and a point of observation belong to the same interface, they are merged into a single test component. This way we assign a tester configuration to each arc of the reduced reachability tree.

Undoubtedly the presented change increases the state space exploration. However it will determine test cases which would not be generated otherwise. As small example of two automata is shown in Figure 3. By application of the two external inputs a and d, one at a time one can generate only the right part of the reachability (solid lines), since the first automaton will go always through state S2 and generate the output o1. Only when the two signals are applied simultaneously the automaton will move to S3 and produce output o2.

In the following section we demonstrate our approach on the example of the foreign agent of the IP Mobility Support.

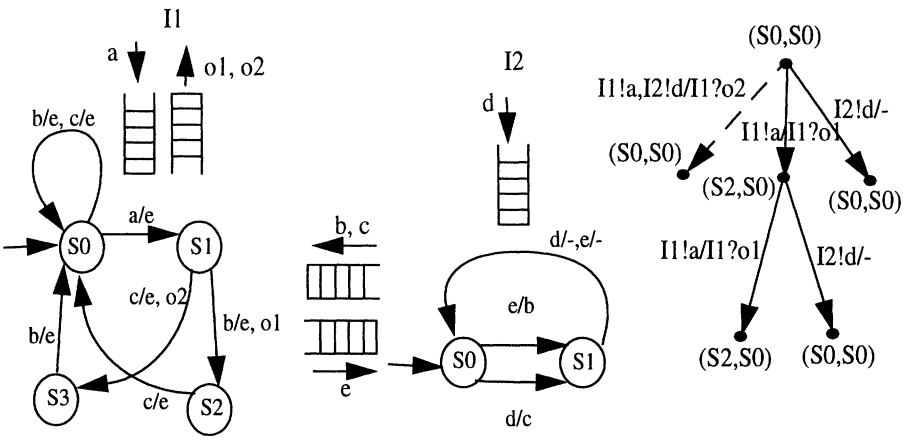


Figure 3 Example CFSM system

6. IP MOBILITY SUPPORT

The IP Mobility Support provides host mobility on the Internet. Accordingly, a mobile host obtains a temporary IP address for the time it visits a foreign subnetwork. This temporary care-of address is used then to re-route packets sent to the mobile host. The mobile host registers its care-of address with a foreign agent of the visited subnetwork and with its home agent. After registration the home agent intercepts packets addressed to the mobile host, and tunnels them to the registered care-of address of the mobile host.

Since the three interoperating components: the mobile host, the foreign agent and the home agent likely to have different ownerships and vendorships we develop a separate test suite for each of them. Here we present the test suite generation only for the foreign agent.

6.1 Foreign Agent

We modeled the case when the foreign agent serves at most one mobile host at a time, i.e. the first registration request is accepted, any other new registration request is rejected until this registration is valid.

The foreign agent is composed of four communicating finite-state machines of Figure 4, two of which are timers and have no external interface (Figure 4b, 4c). The third automaton (Figure 4a) advertises the foreign agent. It generates the appropriate agent advertisement indicating the agents availability. Accordingly the transition between these states are initiated by the

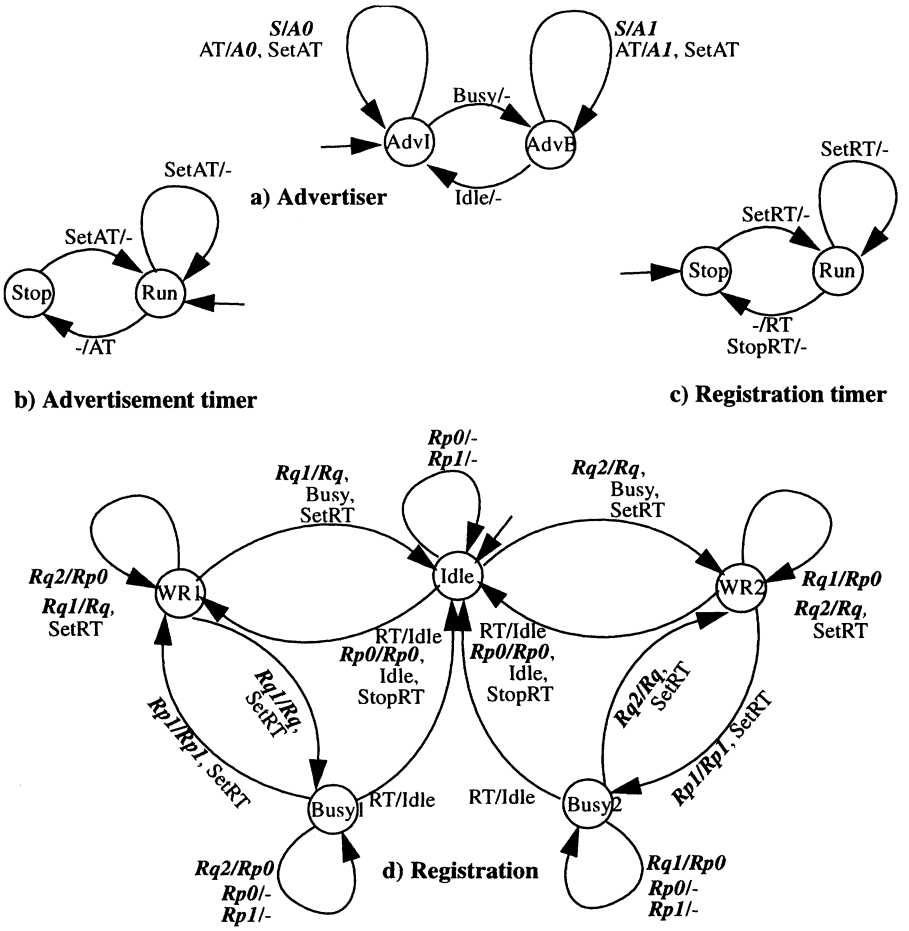


Figure 4 The four automata of the foreign agent

signal of the fourth automaton, which handles the registration (Figure 4d). This automaton becomes busy after forwarding the registration request to the home agent and becomes the end point for tunneling of data packets to the registered mobile host (we neglected the loop transition of forwarding data packets). The automaton distinguishes requests coming from the registered and from a non-registered mobile hosts. We do not model delaying channels for the foreign agent.

Signals occurring at interfaces - both inputs and outputs - are set in bold-italic in Figure 4.

The foreign agent has three interfaces: two toward mobile hosts and one toward a home agent as it is shown in Figure 5. Only the Advertiser and the Registration automata have external inputs, i.e. communicate through these

interfaces, so only their input queues are represented.The matrix of the interface-buffer interconnection is: $R = \begin{bmatrix} \{Rp0, Rp1\} & \{Rq1\} & \{Rq2\} \\ \varnothing & \{S1\} & \{S2\} \end{bmatrix}$

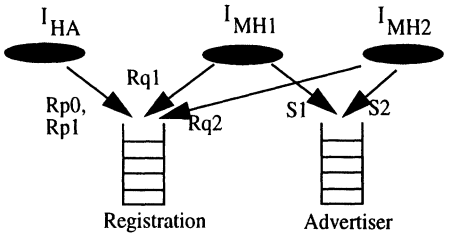


Figure 5 The interface-buffer interconnection in the foreign agent

We do not allow multiple signals at any interface. The matrices of independent signal lists are:

$$R_1 = \begin{bmatrix} \{Rp0, Rp1\} & \varnothing & \varnothing \\ \varnothing & \{S1\} & \varnothing \end{bmatrix}, \quad R_2 = \begin{bmatrix} \{Rp0, Rp1\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{S2\} \end{bmatrix}, \quad R_3 = \begin{bmatrix} \varnothing & \{Rq1\} & \varnothing \\ \varnothing & \varnothing & \{S2\} \end{bmatrix},$$

$$R_4 = \begin{bmatrix} \varnothing & \varnothing & \{Rq2\} \\ \varnothing & \{S1\} & \varnothing \end{bmatrix}, \quad R_5 = \begin{bmatrix} \{Rp0, Rp1\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix}, \quad R_6 = \begin{bmatrix} \varnothing & \{Rq1\} & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix},$$

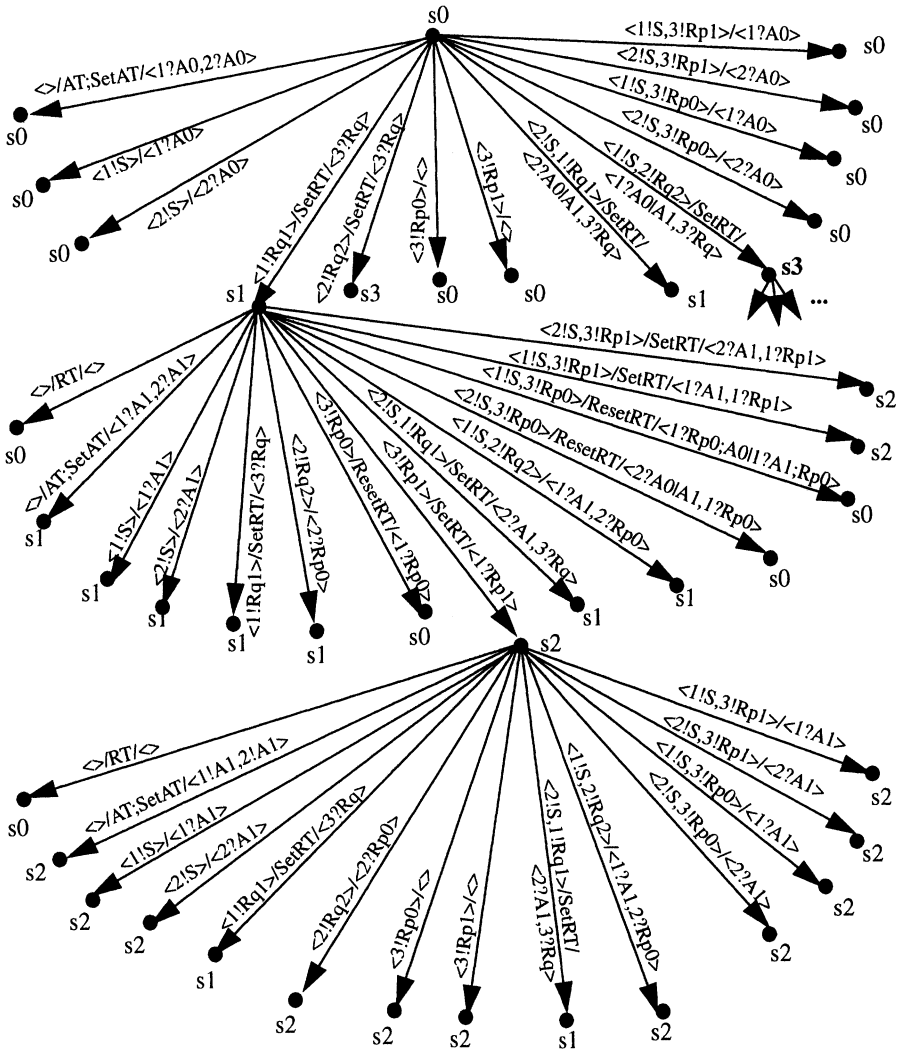
$$R_7 = \begin{bmatrix} \varnothing & \varnothing & \{Rq2\} \\ \varnothing & \varnothing & \varnothing \end{bmatrix}, \quad R_8 = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & \{S1\} & \varnothing \end{bmatrix}, \quad R_9 = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{S2\} \end{bmatrix}, \quad R_{10} = \begin{bmatrix} \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing \end{bmatrix}.$$

We use at most two interfaces in any of these case. The set of composite inputs is: Λ={<*S1, Rp*0>, <*S1, Rp*1>, <*S2, Rp*0>, <*S2, Rp*1>, <*S2, Rq1*>, <*S1, Rq2*>, <*Rp*0>, <*Rp*1>, <*Rq1*>, <*Rq2*>, <*S*1>, <*S2*>, <>}.

The generated reduced reachability tree of the CFSM model with use of the Λ set of composite inputs is shown in Figure 6. The notation used in the reachability tree is the following:

- Nodes represent stable global states of the system. Each of them denote a 4-tuple composed of the states of automata in the form <Advertiser, Registration, Advertisement timer, Registration timer>.
- Arcs have labels of three parts: <inputs> / timer events / <outputs>
  - the set of inputs which initiated the transition, in angle brackets;
  - between slashes internal timer related events if applicable;
  - the generated outputs again in angle brackets.

  Listing the timer events in the graph let us derive timer events for test cases if necessary.



Timers: AT - Advertiser Timer, RT - Registration Timer

Interfaces: 1 - I_{MH1}, 2 - I_{MH2}, 3 - I_{HA}

States: s0 -<AI, I, R, S>, s1 - <AB, WR1, R, R>, s2 - <AB, B1, R, R>,
s3 - <AB, WR2, R, R>, s4 - <AB, B2, R, R>

Advertiser: AI - AdvIdle
AB - AdvBusy

Registration: I - Idle
WR1,2 - WaitForReply1,2
B1,2 - Busy1,2

Timers: S - Stop
R - Run

Figure 6 The reachability tree of the foreign agent

- In front of each input/output event we indicate the interface at which it takes place. Simultaneous input/output events of different interfaces are separated by comma.
- Alternate outputs at the same interface are separated by a vertical bar "|".
- A sequence of output events is concatenated by semicolon ";".

In the reachability tree we indicated but did not expand a subtree rooted in state "s3", which is symmetrical to the subtree rooted in the stable global state "s1". We derived test cases for each arc of the reachability tree, except the subtree rooted in "s3". A certain care had to be taken regarding the timers and the coordination of the test components. One can notice in the reachability tree, that once the foreign agent accepted a registration the only way back to the initial state of the system is the expiration of the registration timer. This means no actions from the tester side, i.e. it has to wait at least for the amount of time the registration was accepted. Therefore these - no input from the tester - branches of the reachability tree were selected "to reset" the IUT to the initial state. 41 test cases and 30 test steps were derived manually from the reachability tree for the MIP foreign agent. 18 of the test cases take into account simultaneous inputs, and they require 16 of the test steps. This means that the application of composite inputs doubled the size of the generated test suite compared to the approach of the strong reasonable environment with a single input signal at a time.

Table 1: Sample test step: Preamble

Name	Preamble (A: PCO, B: PCO)		
Label	Dynamic Behavior	Verdict	prev. notation
	A ! RegistrationRequest, Start (RegistTimer)		Rq
	B ? RegistrationRequest	PASS	Rq
	B ? otherwise	FAIL	

Table 2: Sample test step: Parallel Test Component_1

Name	ParallelTestComponent_1 (A: PCO)		
Label	Dynamic Behavior	Verdict	prev. notation
	A ! RegistrationReply(reject)	(pass)	Rp0

Table 3: Sample test step: Parallel Test Component_2

Name	ParallelTestComponen_2 (A: PCO)		
Label	Dynamic Behavior	Verdict	prev. notation
	A ! Solicitation		S
	A ? RegistrationReply(reject), Cancel (RegistTimer)		Rp0
	A ? AgentAdvertisement(idle)	PASS	A0
	A ? AgentAdvertisement(busy)		A1
	A ? RegistartionReply(reject)Cancel (RegistTimer)	PASS	Rp0
	A ? Otherwise	FAIL	

Table 4: Sample test case

Label	Dynamic Behavior	Verdict	prev. notation
	+ Preamble (MH1, HA)		
	create (PTC_1: ParallelTestComponent_1 (HA) PTC_2: ParallelTestComponent_2 (MH1))		
	[R=pass]	PASS	
	[R=fail]	FAIL	

An example test case presented in Tables 1-4 which was derived from the consecutive arcs: s0 <1!Rq> /SetRT/ <3?Rq> s1 and s1 <1!S, 3!Rp0> / ResetRT/ <1?Rp0;A0 | 1?A1;Rp0> s0. The first arc is used as the preamble for the second one.

The test case specifies two parallel test components: PTC_1 which has a PCO toward the HA(3) interface (Table 2) and PTC_2 which has a PCO connected to the MH1 (1) interface (Table 3). The main test component starts with the preamble then creates the two parallel test components and finally evaluates the verdict (Table 4).We found that testing of this mobile protocol does not differ from stationary protocols except that it requires multiparty testing and the timers play essential role in the protocol, so testing of them is inevitable. We also found that the timer handling is ambiguous in the extension of TTCN in the case of parallel test components: Each newly created test component receives a copy of all running timers, however it is not clear how a timer is stopped if only one of the components detects the appropriate event.

7. CONCLUSION

We approached the problem of test suite generation for composite systems from a structural point of view. We investigate the question of signal dependency to determine the required test method. I.e. we establish, for a system specified in SDL, the sets of signals that can happen simultaneously, therefore need to be offered concurrently to verify the system's conformity. To offer simultaneous signals one has to apply multiparty testing.

In SDL the communication happens via queues buffering, and serializing concurrent signals. Therefore our main criterion of signal independence is being submitted to different buffers. We use the interface-buffer interconnection matrix to determine signal independence. In the matrix signal lists of the same row belong to the same buffer, so only one of them is selected at a time. Additionally one may assume that an interface delivers only a single signal at a time, which restricts further the selection by allowing at most one element in each column. After the selection of independent signal lists we create the set of composite inputs taking at most one signal from each list. Subsequently we use these composite inputs to generate the reduced reachability tree of the system. This approach can be interpreted as an adaptation of the strong reasonable environment reduction technique to multiparty testing.

We have demonstrated this method on the example of the foreign agent part of the IP Mobility Support protocol. The work was done manually with the use of an SDL simulator. The next step will be to implement the algorithm. The critical point is the derivation of the synchronization messages of test components. We found that timer handling of parallel test components is ambiguous in the extension of TTCN. In the MIP protocol due to mobility, timer events could be detected by a different test component than the one, which sent the message setting the timer. At the moment, we had to specify explicitly the synchronization between relevant test components. However this work was very cumbersome and errorprone, at the same time testing of these features for mobile protocols is essential. Therefore an implicit synchronization of timer events between test components provided by TTCN standard would be highly beneficial.

REFERENCES

[CTMF] ISO/IEC 9646 IT-OSI, OSI Conformance Testing Methodology and Framework

[Ek97] A.Ek, J.Grabowski, D.Hogrefe, R.Jerome, B.Koch, M.Schmitt: Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL Specifications, Institute for Telematics, University of Lübeck, Technical Report A-97-03, 1997

[Fern96a] J.C.Fernandez, C.Jard, T.Jéron, L.Nedelka, C.Viho: Using on-the-fly verification techniques for test generation of test suites, INRIA Technical Report No 2987, France, 1996

[Fern96b] J.C.Fernandez, C.Jard, T.Jéron, L.Nedelka, C.Viho: An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology, INRIA Technical Report No 2923, France, 1996

[FMCTa] Revised Working Draft on "Framework: Formal Methods in Conformance Testing", JTC1/SC21/WG1/Project 54.1, January 1995

[FMCTb] Revised Working Draft on "FMCT guidelines on Test Generation Methods from Formal Descriptions", JTC1/SC21/WG1/Project 54.2, February 1995

[Grab96] J.Grabowski, R.Scheurer, D.Toggweiler, D.Hogrefe: Dealing with the complexity of state space exploration algorithms for SDL, Proceedings of the 6th GI/ITG techn. meeting on FDTs for Distributed Systems, University of Erlagen, Germany, 1996

[Grab97] J.Grabowski, R.Scheurer, Z.R.Dai, D.Hogrefe: Applying SAM-STAG to the B-ISDN Protocol SSCOP, Institute for Telematics, University of Lübeck, Technical Report A-97-01, 1997

[Kang97] D.Kang, M.Kim, S.Kang: A Weighted Random Walk Approach for Conformance Testing of a System Specified as Communicating Finite State Machines, FORTE/PSTV'97

[Lee96] D.Lee, K.K.Sabnani, D.M.Kristol, S.Paul: Conformance Testing of Protocols Specified as Communicating Finite State Machines - A Guided Random Walk Based Approach, IEEE Trans on Communications, Vol.44, No. 5, May 1996

[Luo94a] G.Luo, A.Das, G.v.Bochmann: Software Testing Based on SDL Specifications with Save, IEEE Trans. on Software Engineering, Vol. 20, No. 1, 1994

[ObjG] ObjectGEODE on-line documentation, Verilog, 1998

[SDL] ITU-T Z.100, Specification and Description Language (SDL)

[Tan96] Q.M.Tan, A.Petrenko, G.v.Bochmann: A Test Generation Tool for Specifications in the Form of State Machines, ISBN-0-7803-3250-4/96, IEEE 1996

[TTCNe] Amendment 1 to ISO/IEC 9646-3:1992 TTCN extensions