A Pragmatic Approach to Generating Test Sequences for Embedded Systems

Luiz Paula Lima Jr.[1] *and Ana R. Cavalli*
Institut National des Télécommunications
9, rue Charles Fourier - 91011 Evry Cedex - France
Tel: (+33 1) 60 76 44 74 Fax: (+ 33 1) 60 76 47 11
{lima|Ana.Cavalli}@hugo.int-evry.fr

Abstract

Application architectures have evolved to distributed architectures where applications are no longer seen as software blocks, but rather as cooperating software components, possibly distributed over the network. Some of the application's components may have already been thoroughly tested while others have not. This paper presents a pragmatic solution to component testing by means of controlling the composition process in order to identify global transitions that reflect the component's behaviour. The application of the proposed method is illustrated by an example based on the handling of a telephone call.

Keywords

Test generation, component testing, embedded systems, distributed architectures, automata composition.

1. This work was funded by CNPq.

1 INTRODUCTION - COMPLEX SYSTEMS

Application platforms have evolved from monolithic architectures to distributed ones where a system is seen as an open set of interworking components. These systems are complex for their components are usually hierarchically organized and may have a certain degree of autonomy. In these systems, all external events can affect any part of its internal state. This is the primary motivation for vigorous testing, but for all except the most trivial systems, exhaustive testing is impossible [1]. In other words, this increasing complexity of computer systems and their communication protocols can no longer be handled by traditionally informal or *ad hoc* methods for conformance and interoperability testing [2]. Since we have neither the mathematical tools nor the intellectual capacity to model and test the complete behaviour of large discrete systems, we must either be content with acceptable levels of confidence regarding their correctness or try to find out other ways to tackle their complexity.

Abstraction is one of the most prevalent techniques to deal with complexity. In the domain of conformance testing, for instance, a common abstraction is not to consider system's internal signals when generating test sequences. Of course, the choice of the level of abstraction (or what are the primitive components in a system) is relatively arbitrary and is largely up to the discretion of the observer of the system [1].

The application of embedded testing techniques is also an important aspect to consider when simplifying the validation of these systems. But current experience has shown that the embedded nature of components make the current type of automatic test generation useless. This has been the case for the GSM-MAP protocol [3], and for the SSCOP protocol environment for AAL5 (ATM Adaptation Layer) [4]. For instance, the application of these new techniques to the interaction between SSCOP and Q2130 on top, and their relation with the Q2931 signalling protocol (to which they provide the SSCF service), would be of particular interest.

In this paper, we present a pragmatic solution to component testing of complex systems by means of abstraction (defining a composition algorithm that removes internal actions) and by means of controlling this composition process in order to identify global transitions that reflect the behaviour of the component under test. The paper is organized as follows. Section 2 introduces the idea of embedded testing, underlining its relevance in the context of complex systems. Section 3 suggests a test architecture for embedded systems together with basic definitions and assumptions. Our embedded testing techniques are based on an algorithm for automaton composition that is detailed in Section 4. Section 5 presents a method for deriving test sequences for complex systems using, basically, goal-oriented techniques and our tools. Conclusions are drawn in Section 6.

2 EMBEDDED TESTING BASICS

It is widely accepted that testing is a crucial phase in the development of complex systems such as communication protocols [2]. Nevertheless, there is a strong need for systematic methods for testing these systems since the existing methods for test derivation from *Labelled Transition Systems* (LTS) and *Input/Output Finite State Machines* (I/OFSM) (based on the "black-box" representation of the *implementation under test* - IUT) are not adequate in this context. In fact, some of the system components may have already been thoroughly tested or a certain level of confidence may have been assigned to them so that they no longer need to be subject of test. Test derivation methods that generate test sequences for only a subset of the system components are called "*embedded testing methods*" [5] or "*gray-box testing methods*" [2] or even "*methods for testing in context*." [6]

Example 1. Consider the system depicted in Figure 1. Assume that module *C* is known to be faultless and that module *I* must be tested[1]. Let us also assume that we do not have access to *I*'s internal interfaces (since the implementation of the system is given as a black box). Therefore, internal signals sent to/from *I* may reach the environment after passing through *C*. Module *C* acts as a kind of "filter" and system responses to environment stimuli must be correctly interpreted in order to verify that module *I* works as specified. ❑

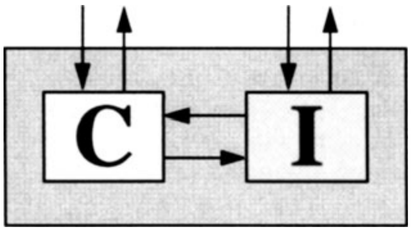

FIGURE 1. A generic complex system.

Traditional methods for testing in isolation turn out to be inadequate for, basically, two reasons:

- Module *I* can neither be "removed" from the system (in order to be tested in isolation) nor can it give access to its internal interfaces. Traditional methods are then obliged to test the system as a whole.
- Obviously, testing the whole system would test module *I* as well, but then we would have (unnecessarily) tested a part of *C*'s behaviour that is independent of *I*. This happens, because the system's global behaviour is likely to contain behaviour that only concerns module *C*.

Embedded testing represents situations that occur very frequently in protocol conformance testing, functional testing of digital circuits (specially, multiprocessor networks) as well as in testing of object-oriented packages[2]. Although the details of each component implementation may remain hidden, to be able to test such sys-

1. Modules C and I may be viewed as the composition of all machines of the system that are not under test and that are subject of testing, respectively.

tems, we must have information about the component configuration (or structure) within the system. Embedded testing methods take advantage of the information about the configuration of the complex system components.

To sum up, embedded testing is concerned with testing a system component when the tester does not have direct access to its interfaces. The access is then made through another process which acts as a sort of "filter." According to [5], if "control and observation are applied through one or more OSI implementations which are above the protocol(s) to be tested, the [testing] methods are called embedded."

Pragmatic embedded testing techniques identify parts of the system's global behaviour that reflect the behaviour of the component under test, and then performs tests on only those parts. Intuitively, the set of test sequences to test the whole system contains redundant or unnecessary elements that we would like to avoid when testing.

3 TEST ARCHITECTURE FOR EMBEDDED SYSTEMS

3.1 Preliminary Definitions

An *input-output finite state machine (I/OFSM)* is a tuple: $\langle S, I, O, \delta, \lambda, s_0\rangle$ where S, I, O are finite, non-empty sets of states, inputs and outputs respectively; s_0 is the initial state; $\delta : S \times I \rightarrow 2^S$ is the state transition function and $\lambda : S \times I \rightarrow 2^O$ is the output function.

One test is referred to as a *test case*. A *test suite* is a set of test cases that tests all conformance requirements.

The action of the conceptual tester involves interactions with the *System Under Test (SUT)*. These can, in theory, be observed and controlled from several different points that are called *Points of Control and Observation* (*PCOs*). PCOs can be modelled as two queues: an output queue for control of test events to be sent to the Implementation Under Test (IUT); and an input queue for the observation of test events received from the IUT.

For testing purposes, a complex system may be divided in two subsystems: a non-empty set of *components under test* (or simply *component*, or *IUT*), and a (possibly empty) set of components that are not concerned by testing (the *context*). The problem of testing a system with an empty context reduces to the traditional problem of testing in isolation.

2. In this work, we are particularly interested in ODP-like systems where different objects communicate within an arbitrary configuration and where we do not intend to test the entire system, but only some of its components.

3.2 Architecture

The *test architecture* is the description of the environment in which the component is tested. It describes the relevant aspects of how the component is embedded in other systems during the testing process, and how it communicates via these embedding systems with the tester (see Figure 2).

A test architecture consists of [5]:

- a tester;
- an implementation under test (IUT);
- a test context;
- points of control and observation (PCOs);
- implementation access points (IAPs, also "interfaces").

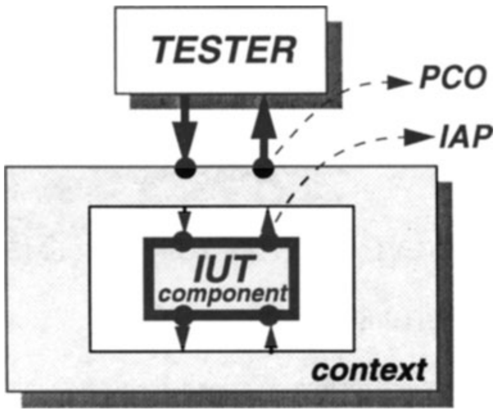

FIGURE 2. Generic test architecture.

In the ideal test architecture (for testing in isolation), the IAPs and the PCOs coincide, the test context is empty, and the tester interacts directly with the IUT. This is rarely the case in real systems, though. The *System Under Test* (SUT) is composed of the IUT and the test context.

The tester is equipped with a timer that is started when a signal is sent to the SUT. On receipt of a response from the SUT, this signal is checked with respect to the test case. After a time out period, if no signal is received, then a fail verdict is issued. Input data for the tester consists of the test suite which guides all testing activities expressing what signals should be sent to the SUT, and what the expected responses are. The test suite represents the reference system in the tester.

3.3 Hypothesis

In order to be able to employ the embedded method for test derivation, described in Section 5, we make the assumption that the context is correctly implemented and that a faulty component implementation does not increase the number of states of the global machine. The latter is a variation on a common hypothesis for testing in isolation [7] that makes it possible to evaluate the test coverage in embedded testing.

The IUT interacts with its context through synchronous communication with input queues of finite size. This implies that a next input x is only submitted to the system after it has produced an external output y in response to the previous input (*I/O ordering constraint* [6]).

The SUT is "reactive" in the sense that one input signal can trigger one or more outputs which are simultaneously sent back to the environment. That is, an output (or set of outputs) must be identified as a response of the system to a particular given stimulus or input.

4 TRACE-KEEPING ALGORITHM FOR AUTOMATON COMPOSITION

The generation of test sequences from formal specifications of systems has been traditionally based on the exhaustive simulation of the specification in order to obtain an automaton that represents the global behaviour of the system. Since it is impossible, in most of the cases, to deal with the size of the automaton that represents the complete behaviour of these systems, a reasonable approach is to simulate the execution of the specification by controlling the range of values assigned to each internal variable and each parameter of input messages. The closer this range is to the real one, the more realistic and the larger the test will be. Obviously, there is always a compromise between accuracy (completeness) of the automaton and its size. But, even with an automaton of a "computable" size, the process of test sequence derivation may not be able to cope with that automaton in a reasonable period of time.

To date, to generate test sequences, what we have done is to take the "big" automaton (that is, the one which is as close to the specification as possible) and then, through the definition of view points (PCOs), abstract the signals which are irrelevant in the current consideration or view point. Then, we may proceed by minimizing the automaton using an algorithm (described in [3]) which removes all internal signals (if the choice of the PCOs is well done). We thereby obtain an automaton that corresponds to the "big one," but abstracting details we do not yet want to consider (see Figure 3a). In general, this automaton has a reasonable size, and therefore it can be used as input for the process of deriving test sequences.

However, even with a "big" automaton generated by simulation, the "reduced" one is often simpler than we would like. Producing an even "bigger" automaton would, in principle, result in a bigger "reduced" automaton, but in many cases a "bigger" automaton just cannot be generated due to storage, memory or computational limitations.

To solve these problems we will consider the use of *composition algorithms* (see Figure 3b). The idea is to avoid the initial automaton size explosion by dividing (which is often already done, if we are dealing with modular systems) the specification into smaller, interrelated modules which are then simulated to produce more complete or smaller automata. The simple composition of these automata, without taking into account any kind of abstraction (PCOs), would lead to the "big" automaton of the traditional case that corresponds to the Cartesian product of the two automata. However, if we use information about our abstraction level we are able to compose them and at the same time avoid the explosion of the model. In other

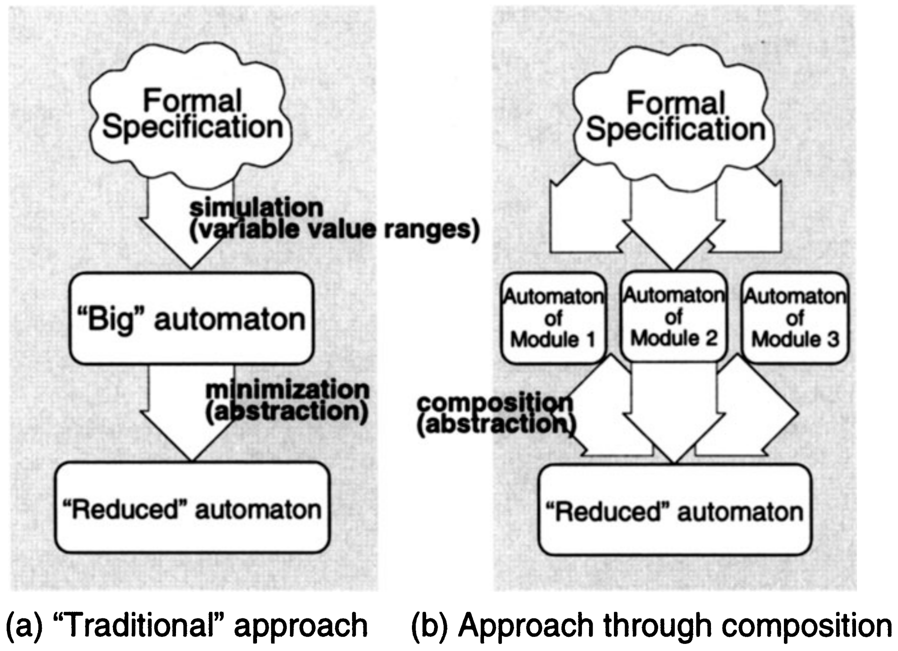

FIGURE 3. Approaches to produce the reduced automaton that will be used as input to the test derivation process.

words, we compose the automata removing internal signals which are not part of what we want to consider for the moment. Composition is done through simulation in order to avoid the generation of unreachable states.

4.1 Definitions

Before proceeding, let us define some useful terms. Let A_1 and A_2 be two I/OFSMs and Π be the cartesian product of A_1 and A_2.

Definition 1: A *global state* is a state of Π. The set of all global states is denoted by $\Gamma = \{\sigma | \sigma \text{ is a state of } \Pi\}$.

Definition 2: A *reachable global state* is a global state that is attained during the joint execution of A_1 and A_2. The set of all reachable global states is denoted by
$\Re = \{\rho | (\rho \in \Gamma, \rho \text{ is attained during the joint execution of A1 and A2})\}$.

Definition 3: An *unreachable global state* is a global state that never happens in the joint execution of two machines.

Definition 4: A *stable global state* is the global state that Π reaches after sending a response to environment and before receiving another signal from it. Let us denote the set of all stable global states by

$\Sigma = \{\tau | (\tau \in \Re, \tau \text{ is attained just after sending a signal to the environment})\}$

Definition 5: A *transient global state* is a reachable state which is not stable. Many internal message exchanges and state changes can take place after receiving a signal from the environment and before sending back a response to it. These intermediary states are called transient global states.

The relation between Γ, $\Re$ and Σ is given in Figure 4. The set of transient global states is given by $\Re - \Sigma$.

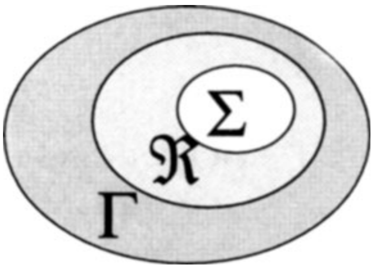

FIGURE 4. Relation between sets of states.

Example 2. Let us consider the automata depicted in Figure 5. Using composition without consideration of internal signals, we would obtain an automaton that is the cartesian product of the two first automata (Figure 6). Internal signals in the cartesian product machine can be hidden using algorithms like the one describe in [3].

Considering *ia* and *ib* internal signals, the automaton that corresponds to the global behaviour, after hiding these signals is depicted in Figure 7.

However, if we compose both automata already taking into account information about the internal signals, we will obtain the same result with the advantage of not producing the intermediary large automaton which corresponds to the cartesian product.

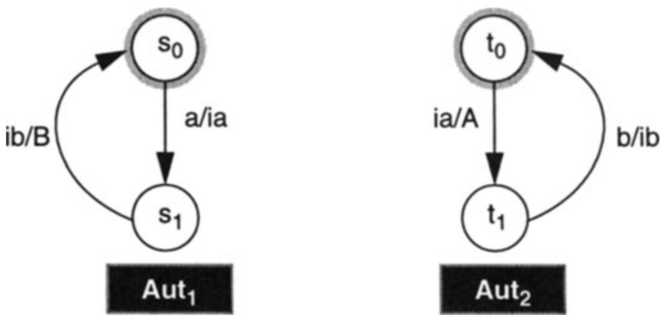

FIGURE 5. Simple example for automata composition.

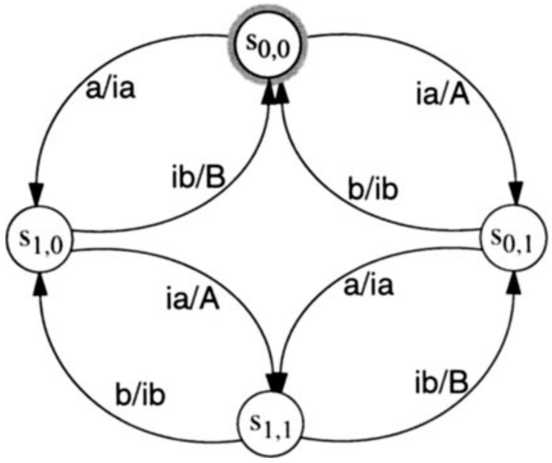

FIGURE 6. Cartesian product of Aut$_1$ and Aut$_2$.

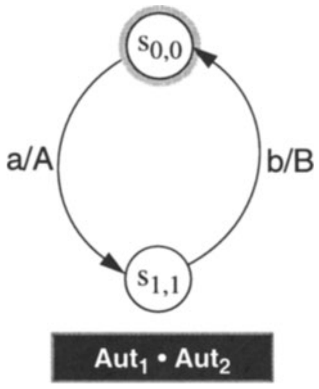

FIGURE 7. Automaton representing the joint external behaviour of Aut$_1$ and Aut$_2$.

In this example, $\Gamma = \{s_{0,0}, s_{1,0}, s_{1,1}, s_{0,1}\}$, $\Re = \{s_{0,0}, s_{1,0}, s_{1,1}\}$, and $\Sigma = \{s_{0,0}, s_{1,1}\}$. There is only one transient state ($s_{1,0}$) and one unreachable state ($s_{0,1}$). ❑

4.2 Composition Algorithm

In this section we describe an algorithm to compute the global composition of two automata while removing internal actions. The algorithm is made as modular as possible, so it may be implemented in a distributed fashion. (Our current implementation is centralized, however it serves our purpose.)

4.2.1 Input data and object configuration

Let A_1 and A_2 be the two automata we intend to compose, let **E** be the set of the signals exchanged with the environment[1] and $k \in \{1, 2\}$.

The diagram of Figure 8 shows the object configuration used in the composition

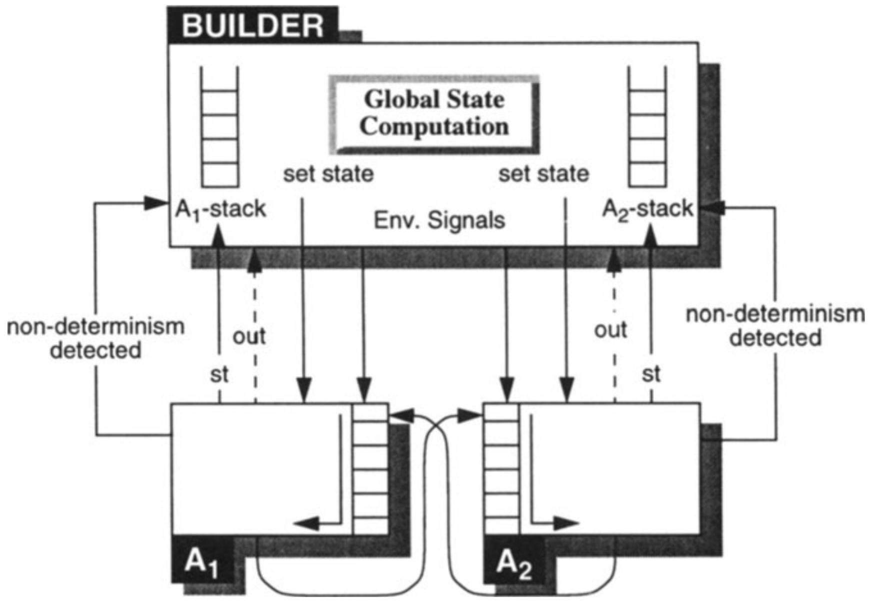

FIGURE 8. Object configuration in the automata composition algorithm.

process. There are basically three objects that communicate by means of message passing: *objects* A_k, and the *builder.*

An A_k *objects* implements automata behaviour. Incoming signals are placed in an input queue and consumed as soon as possible. They cause an outgoing transition

1. E corresponds to the PCO definition.

from the current state to be traversed producing an output either to the *builder* object or a peer object (A_k). Also, each state change is reported back to the builder that records them in individual stacks, so it will be able to keep track of all reachable global states during subsequent steps.

The *builder* is the object that controls the composition process and gathers results from objects A_k. These results are used to build up a composite transition, say *trc*, which is instantiated at the end of each step. Getting an output signal from either A_k means that it has obtained all the necessary information to instantiate *trc* and that it can advance to the next step.

4.2.2 The Composition Process

Initially, A_1 and A_2 are set to their initial states which correspond to the global initial state. The builder is aware of which signals could be processed by each machine in each state. It then sends a signal to, say A1, and waits for a response from either A_1 or A_2. Meanwhile, many massage exchanges may take place between A_1 and A_2 until they reach a stable state (when their input queues are both empty and an external signal is sent back to the builder). In order to compute subsequent global states, all reachable states must be saved by the builder in its stacks[1]. A global transition is then instantiated from:

- A_1 and A_2's initial states (before sending the signal);
- the signal sent to the system;
- the system's response; and
- the composite stable state that is composed of the states reached by each machine.

This procedure is repeated until there are no more unvisited outgoing transitions from the current global state whose inputs belong to E.

Upon receipt of a signal, each A_k object changes its internal state and sends a signal to another object (a peer object or the builder).

This approach differs from the synchronous product described in [8] and [9]. In fact, while in the synchronous product a transition belongs to the product machine if it can be traversed in the two components or if it can be traversed only in the specification [9], in our composition algorithm, each environment signal is sent to and received from either the context or the component machine, and what is modelled is their joint execution with internal signals being exchanged between them. However, the algorithm of Section 4.2 can be used to obtain the same result as the synchronous product, composing an artificial context that makes visible only some parts of

1. All reachable states are potentially stable states (reachable states may be transient or stable, according to Section 4.1). That is why they must be saved in the builder, so that the builder will be able to get back to them later on.

the component behaviour. An additional advantage over the synchronous product is better control over the observation (i.e. only input - or output - sequences may be observed, if desired).

4.2.3 Extensions - Transition Marking and Behaviour Exploration

The algorithm also includes a complex scheme of transition marking, is also needed to tackle the following issues:

- Multiple (simultaneous) outputs (i.e. when one signal from the environment stimulates several simultaneous outputs);
- Live-lock detection (if components exchange messages indefinitely);
- Simultaneous triggering of multiple transitions (with simultaneous state changes).

If the machines are non-deterministic, then a mechanism of behaviour exploration guarantees that all possible branches are examined. A_1 or A_2 warn the builder when there is a non-deterministic choice for the last input, so that the builder will send the same signal a second time and a different transition will then be traversed. As a result, non-deterministic machines are usually produced.

4.2.4 Errors and Warning Messages

There are basically two undesired situations that may happen during the composition process and that are reported back as errors or warnings:

1. **Incompatibility errors:** A_k was not expecting a given internal signal from its peer machine at its current state. In this case, the internal signal is simply "forwarded" to the environment (builder) that instantiates a global transition with an error message (for it contains an internal output signal).
2. **Unreachability warnings:** During the joint execution of both machines some transitions of either machine may not be traversed and some states even may not be visited. This means that a part of the machine behaviour was not exercised in the joint execution. This kind of information can be useful, for instance, for feature interaction detection [10].

In the first case, either the machines were not designed to work together or they are badly specified. In the second case, however, there may be represented situations where the component presents (additional) functionalities that are not used by its context (or vice-versa).

4.3 Example: *Subscriber Connection Unit (SCU)* and *Subscriber*

Let us use the described algorithm to compose the I/OFSMs presented in Figure 9 (internal signals are underlined in both automata). These machines represent the behaviour of a telecommunication system that is composed of two processes: the *Subscriber* and the *Subscriber Connection Unit.* They specify the handling of the arrival of a telephone call and are composed of states whose names are given in Table 1.

TABLE 1. State names for SCU and Subscriber.

SCU		Subscriber	
State number	**State name**	**State number**	**State name**
S_0	idle	T_0	idle
S_1	wait_for_answer	T_1	ringing
S_2	conversation	T_2	wait_for_stop_ringing
S_3	control_by_called	T_3	conversation
S_4	fault	T_4	control_by_called
		T_5	fault

The composition algorithm proceeds as follows: the builder sets both machines to their initial states (assume that the initial global state is S_0T_0). From these states, there is only one external signal that can be treated by the SCU, namely, *call_arriving* (*call* is an internal signal). Because there is a non-deterministic choice for signal *call_arriving*, assume it traverses transition [S_0,"call_arriving/call",S_1]. Since the output *call* is an internal signal, it is sent to the Subscriber causing a state change (from T_0 to T_1) and a *NULL* signal to be sent back to the builder. Upon receipt of a signal from the Subscriber, the builder understands that the system has reached a stable state and that a new global transition can be instantiated (in this case, [S_0T_0,"call_arriving/NULL",S_1T_1]). A new reachable global state S_1T_1 is saved in the builder for later analysis.

Since there is another non-traversed transition from state S_0T_0 with an external input (transition [S_0,"call_arriving/NULL",S_0]), both machines are reset to their respective states (S_0 and T_0) and *call_arriving* is sent again to the SCU which now traverses transition [S_0,"call_arriving/NULL",S_0], and another global transition ([S_0T_0,"call_arriving/NULL",S_0T_0]) is instantiated.

Since there is no other non-traversed transition from state S_0T_0 with an external input, a new global state is computed from the set of reachable global states (S_1T_1)

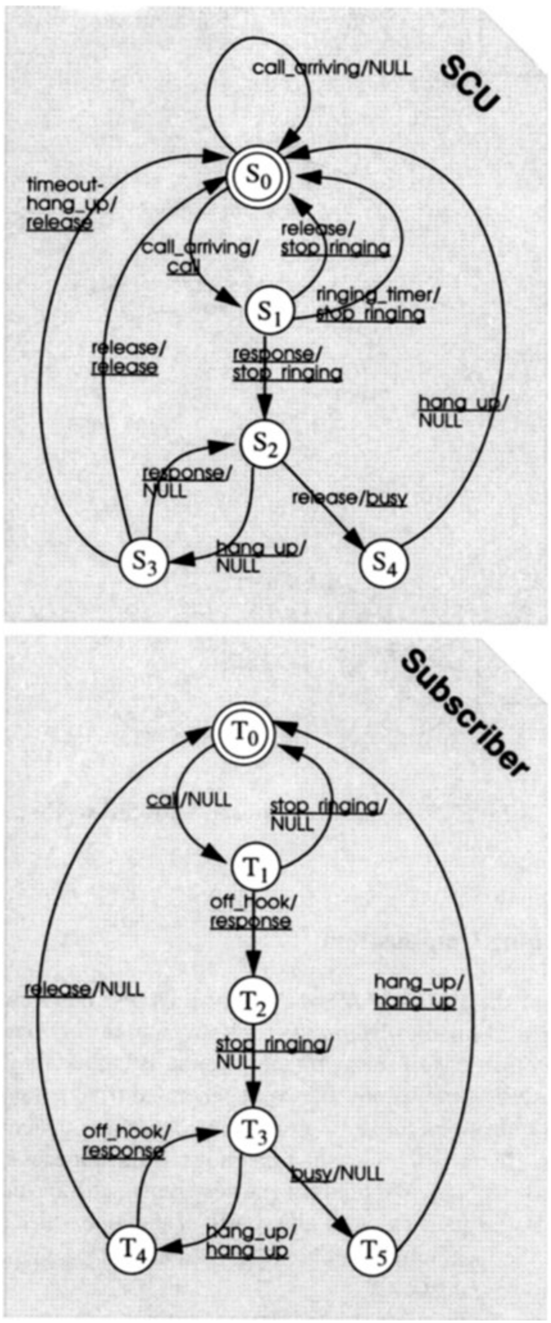

FIGURE 9. Input automata used as input examples for the composition algorithm.

and the process continues until no other global state can be obtained (the set is empty).

The composite automaton obtained is depicted in Figure 10.

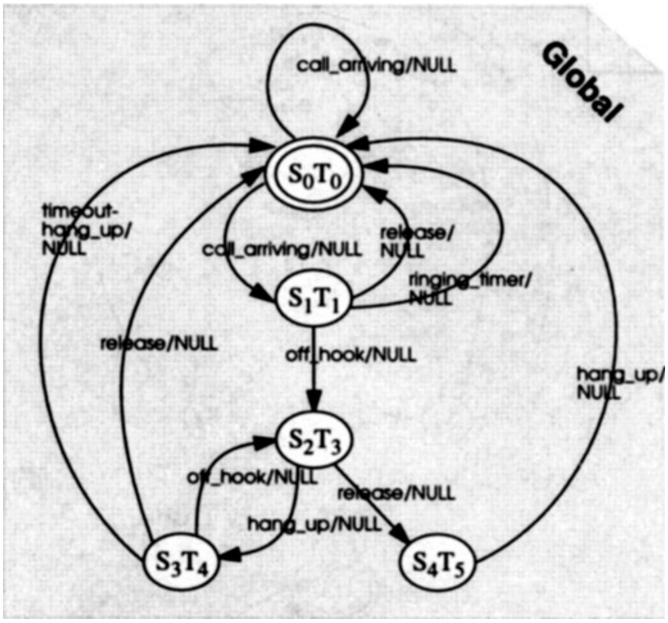

FIGURE 10. UCS composed with Subscriber.

4.4 Trace-keeping Composition

Each transition in the global I/OFSM (i.e. the I/OFSM that describes the global behaviour of the SUT) comes from either a transition of only one component or a combination of transitions of the two components. It is therefore possible to keep track of global I/OFSM transitions that were generated from a transition of the IUT, and to use these transitions for testing only the component under test. Doing so, it becomes easy to distinguish relevant transitions from unnecessary or redundant ones in the global machine. Actually, local test sequences are not "translated" in terms of global test sequences, but rather parts of the global behaviour that reflect the behaviour of the local transitions are identified and test sequences are generated for only those global transitions.

In order to better understand trace-keeping composition, we introduce the concept of equivalence in context as defined in [6].

Definition 6: Let "•" represent the composition operation as described in

Section 4.2. Two machines M_1 and M_2 are *equivalent* in a context C if and only if the joint execution of M_1 and C does not contain live-locks (i.e. the composite machine $M_1 \bullet C$ exists); and $M_1 \bullet C$ is equivalent to $M_2 \bullet C$.

An important question at this point is whether testing a global transition is equivalent to testing a corresponding transition of the component machine. The answer is not straight forward. Let C be the context machine, $Spec$ be the component specification and Imp the component implementation. Assume that global transition t ($t \in C \bullet Spec$) was generated by the composition of t_C belonging to the context C and t_S belonging to the component machine $Spec$ (other cases where many implementation/context transitions originate a single global transition are analogous). The absence of transition t in the global machine $C \bullet Imp$ means that the implementation is faulty (since the context is correctly implemented - see Section 3.3). However, if transition $t \in C \bullet Imp$, then either $t_S \in Imp$ or the implementation did something which is equivalent in the context[1].

Example 3. Consider the I/OFSMs depicted in Figure 11 (internal signals are underlined). Although *Imp* is generally considered to be a faulty implementation of *Spec*, it is not, actually, in the context of *C*, because the composition $C \bullet Spec$ is equivalent to $C \bullet Imp$. Therefore, if the global transition labelled *a/d* exists in the composite machine, we cannot affirm that the transition labelled *ia/id* belongs to the implementation. Nevertheless, we are still able to state whether the implementation has a set of equivalent transitions in that context. ❑

This observation leads us to the following conclusion: if there are at least two different paths composed of transitions labelled *internal/internal* (an internal input and an internal output) that lead to the same state in the context machine, then, intuitively, the implementation is free to take the path it wants without changing the aspect of the global behaviour (since all message exchanges are internal). Otherwise, the implementation would be obliged to take the unique existing path in order to preserve the global behaviour.

1. We do not consider here the problem of *latent faults* pointed out in [6], since our testing methods apply a signature to the arrival state of the transition in order to check its correctness.

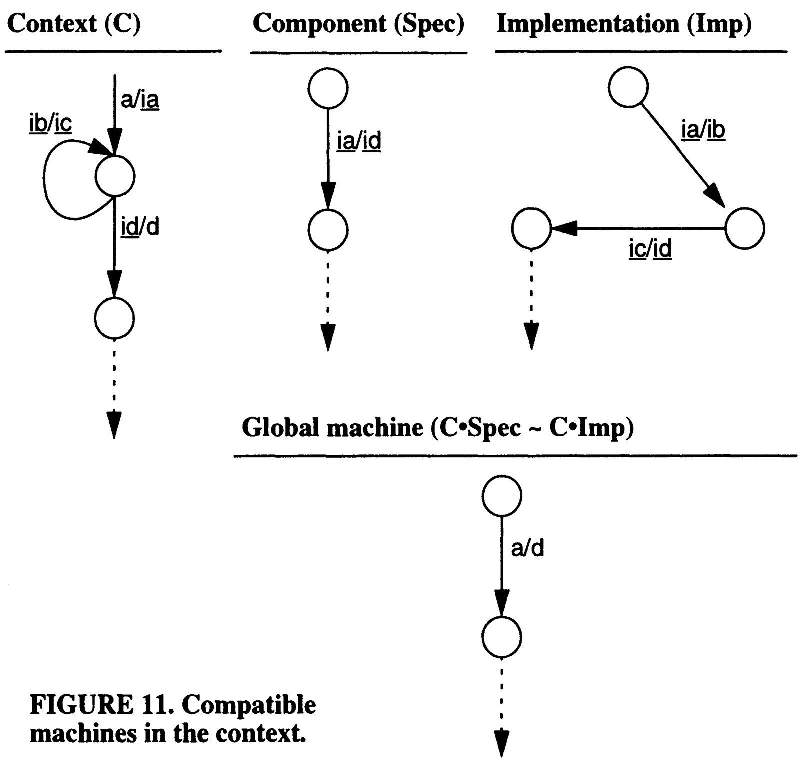

FIGURE 11. Compatible machines in the context.

Example 4. If we consider the Subscriber from the example of Section 4.3 to be our component under test, we observe that,

... in order to test component transition...	**... we should test global transition...**
(T_0,"call/NULL",T_1)	(S_0T_0,"call_arriving/NULL",S_1T_1)
(T_1,"stop_ringing/NULL",T_0)	(S_1T_1,"release/NULL",S_0T_0) *and* (S_1T_1,"ringing_timer/NULL",S_0T_0)
(T_1,"off_hook/response",T_2) *and* (T_2,"stop_ringing/NULL",T_3)	(S_1T_1,"off_hook/NULL",S_2T_3)
(T_3,"busy/NULL",T_5)	(S_2T_3,"release/NULL",S_4T_5)
(T_3,"hang_up/hang_up",T_4)	(S_2T_3,"hang_up/NULL",S_3T_4)
(T_5,"hang_up/hang_up",T_0)	(S_4T_5,"hang_up/NULL",S_0T_0)

... in order to test component transition...	**... we should test global transition...**
(T_4,"release/NULL",T_0)	(S_3T_4,"timeout-hang_up/NULL",S_0T_0) *and* (S_3T_4,"release/NULL",S_0T_0)
(T_4,"off_hook/response",T_3)	(S_3T_4,"off_hook/NULL",S_2T_3)

This is true since there are no alternative paths in SCU whose transitions are all labelled *internal/internal* and provided that the context (SCU) is correctly implemented, which is one of our assumptions (Section 3.3). ❑

5 TEST SEQUENCE GENERATION FOR EMBEDDED SYSTEMS

Goal-oriented testing techniques consist of selecting a subset of the global system's behaviour that is likely to be faulty or that is critical within the system and generating test sequences for only those parts. In general, this selection is made in an *ad hoc* manner by human experts that identify the portions of the system's behaviour that might be subject of testing. Obviously, the system is only partially tested and this technique guarantees a behaviour coverage with regard to the subsystem investigated [8].

In this section, the idea is basically to couple together goal-oriented techniques and trace-keeping composition in order to generate test sequences that concern only the component under test. Using the trace-keeping algorithm of Section 4.2 we can automatically identify the parts of the system that reflect the component's behaviour following which we can use goal-oriented techniques to test this subsystem.
In a non-optimized test generation method each transition is tested in the following manner:

1. Use the shortest path to set the system to the initial state of the transition at hand;
2. Send an input signal and check system's output;
3. Check if the system moved to the correct state.

Many techniques to improve this method have been suggested in the literature and they basically consist of finding a path including all system transitions (in the traditional approach). Since we do not need to test all system transitions, we would be glad to find a path traversing only global transitions that affect the component under test. However, the set of transitions that reflect the component's behaviour in the global machine may not form a (strongly) connected I/OFSM. Therefore, some global transitions that do not concern the component itself may have to be kept when generating the test sequences (this is a problem for goal-oriented testing techniques in general).

We are currently working on a tool called TESTGEN developed at INT in order to incorporate test generation for embedded components. It uses the I/OFSM of the global system (without the internal actions) and a list of transitions to be tested as input data, and it generates test sequences for only the transitions belonging to that list. The tests are performed in two different ways: 1) by defining a tour that starts and ends at the initial state and includes the transitions that define the test purposes (transitions that concern the component under test) or 2) by defining a tour that includes these transitions but also the signatures of the arriving states. In the first case, test sequences are shorter but only detect output faults. In the second case, we are able to detect output and transfer faults. Both are optimized.

6 CONCLUSION

In this paper we have presented a pragmatic approach to generating test sequences for embedded components of complex systems. The approach proposed is based on: 1) the definition of a composition procedure that allows the abstraction of the internal signals exchanged between the processes that compose the system, whilst preserving the exchanges between the system and its environment. The trace-keeping composition algorithm that was defined allows the identification of parts of the global system specification that reflect the component's behaviour; 2) goal-oriented testing. The transitions that reflect the component's behaviour specification can be used to build up test objectives that only test the component's implementation.

This approach presents the following advantages: it is not necessary to test the system as a whole (as is the case for traditional methods); it is possible to test the component's behaviour in context and to detect if the component's implementation conforms to its specification. It is also possible to detect if the system implementation includes an embedded component that is equivalent in context to the component specification.

7 REFERENCES

[1] G. Booch, *Object Oriented Analysis and Design with Applications,* 2nd Edition The Benjamin/Cummings Publishing Company, 1994.

[2] PT, *Component Testing for Mobile and Broadband Telecommunications - COIMBRA,* COPERNICUS Project Proposal, Feb. 1996.

[3] R. Anido, A. Cavalli, T. Macavei, L. P. Lima, M. Clatin, M. Phalippou, *Testing a real protocol with the aid of verification techniques,* XXII SEMISH - Brazil, Aug. 1996, pp. 237-248.

[4] A. Cavalli, B. Lee and T. Macavei, *Test generation for the SSCOP-ATM networks protocol,* SDL Forum'97, September 1997, France.

[5] ISO, *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework,* International Standard IS-9646. ISO, 1991.

[6] A. Petrenko, N. Yevtushenko, G. v. Bochman, *Fault models for testing in context,* Proceedings of FORTE - Kaiserslatern, Germany - 8-11 Oct. 96.

[7] O. Charles, R. Groz, *Formalisation d'Hypothèses pour l'Évaluation de la Couverture de Test,* Proceedings of CFIP'96 - Rabat, Morocco, 14-17 Oct. 1996.

[8] Algayres, B; Lejeune, Y. and Hugonnet, F. *GOAL: Observing SDL Behaviours with GEODE,* Proceedings of the 7th SDL Forum, Oslo, Norway, 26-29 Sept. 1995 - pp. 223-230.

[9] J-C Fernandez, C. Jard, T. Jéron, G. Viho, *Using on-the-fly verification techniques for the generation of test suites,* Summer School MOVEP'96, Nantes, France, 18-21 Jun. 1996.

[10] L. P. Lima, A. Cavalli, *Service Validation - An Embedded Testing Approach,* Proceedings of EUNICE Summer School - Lausanne, Switzerland, 23-27 Jun. 1996.

8 BIOGRAPHY

Luiz Paula Lima Jr. is currently a PhD student at INT (*Institut National des Télécommunications*), Evry, France and he received his MSC degree in 1994 at UNICAMP (State University of Campinas), Brazil. His current research interests include object-oriented distributed systems and platforms (ODP/CORBA) and testing methods for these architectures.

Ana Rosa Cavalli received the Doctorat d'État es Mathematics and Computer Science in 1984 from the University of Paris VII, Paris, France. From 1985 to 1990, she was a staff research member at the CNET (*Centre National d'Etudes des Télécommunications*), where she worked on software engineering and formal description techniques. Since 1990, she joined the INT (*Institut National des Télécommunications*) as professor. Her research interests include formal description techniques, validation of protocols and services, computing methodology and testing methods for distributed architectures.