STUDIA UNIV. "BABEŞ-BOLYAI", INFORMATICA, Volume XLIII, Number 1, 1998

USING tMSC FOR CONFORMANCE TESTING. THE TESTPLAYER APPROACH

IULIAN OBER*

Abstract. The validation phase for a concurrent system is complex and time consuming. The ISO 9646 norm defines a standard framework and language (TTCN) for describing abstract test suites for communication protocols and other reactive systems. TTCN is often viewed as too cryptic, hard to read and poorly adapted for systems other than OSI protocols. This paper presents an alternative approach for executing conformance tests, based on the MSC graphical language complemented by a proprietary imperative language for describing test cases (TDL). Our technique is implemented in a lightweight and open toolbox usable for testing a large class of reactive systems.

1. Introduction

The current state of the art in conformance testing is developed around the ISO 9646 standard Conformance Testing Methodology and Framework [6]. The meaning of conformance testing is black-box testing where the system and the environment interact through messages. Besides the framework, this standard defines also a language, TTCN (Tree and Tabular Combined Notation) for completely describing abstract test suites [5].

One critic, often raised with respect to the current testing methodology and framework, is that it is too complex. Indeed, the TTCN language defines tens of table formats for expressing the test suite structure, the variables and constants, the data types, the Protocol Data Units, the Abstract Service Primitives, the constraints and finally the behavior. The tools for executing tests starting from TTCN descriptions are not always easy to deploy.

To meet the need for a simpler yet not less powerful method for describing and executing conformance tests, we propose an alternative approach: the use of MSC for describing test cases, complemented with a second-level formalism – the Test Description Language (TDL), all within a lightweight and open framework designed for executing tests on a wide class of possible systems. The MSC [3] is a language for expressing execution traces that has been used for a long time for capturing requirements in telecommunication systems design but also as a basis for system simulation and validation, interface specification, etc. Protocol norms often come with requirements, examples or even test cases described with MSC. MSC is standard, mature and graphical, usable for expressing test cases at a first level, and its simplicity appeals.

This paper advocates our testing framework, by showing the strong and weak points of MSC when expressing tests, by proving that they can be overcome by using a second-level formalism for expressing test cases (the Test Description Language) and by describing the architecture of a tool implementing our testing paradigm and emphasizing on the open parts which make the tool usable in many different contexts.

Related work: After its standardization in 1992, MSC gained popularity and research work was put into using it for specifying test purposes and test cases [8,9,10]. This work was based on the parallel efforts for standardizing the formal semantics for the language [4,11]. Other solutions for testing are based on other proprietary formalisms (it is the case of the ATTOL toolbox [1]) or on TTCN.

This paper is structured as follows: section 2 presents the fashion in which we use MSC for expressing test cases. Section 3 introduces the TDL language, the complement of MSC for describing test cases in TESTPLAYER. Section 4 describes the generic architecture of our testing toolbox. Finally, section 5 shows the concrete architecture of our tool in a specific context and gives an usage example.

2. Using MSCs for expressing test cases in TestPlayer

MSC is a widely recognized and used language for representing execution traces for systems where the exchange of messages is the main observable behavior. It is being standardized by the International Telecommunication Union as Recommendation Z.120 [3]. The standardization work began in 1990 and successive versions made the language more and more stable, rich in constructs, formal and mature.

Being a visual formalism, MSCs are more compact and readable, which makes them more attractive to the users. As a consequence, there were early attempts to use MSC for expressing test cases or test purposes [8,9,10]. At a first look, MSCs are suitable for this purpose, considering that the behavioral part of a TTCN test case describes also a set of possible traces of messages exchanged between the system and its environment, each such trace having associated a verdict at the end. Anyway, several weaknesses of MSCs when expressing test cases were outlined by these early attempts: there is no notion of test verdict in MSC, no possibility to specify types and TTCN-like constraints, no possibility to use variables and no notion of test architecture [7]. We will see in what follows how we overcame these problems in our approach with TESTPLAYER.

2.1 Interpreting test MSCs

The first step in our approach is to give an intuitive meaning to MSCs in the context of testing. This means giving a set of constraints that must be satisfied by the test MSCs (denoted tMSC in what follows) as well as an interpretation to each construct that may appear on such an MSC. There are three flavors of MSCs that may be used with TESTPLAYER, differentiated by the instances that they represent (see Figure 1).

The first one, which may be called pure tMSC taking the point of view of the tester, may contain instances representing the PCOs (Points of Control and Observation – ISO 9646 terminology [6]) and instances representing the testers (one or more). We will use this flavor of tMSC when detailing the meaning that we give to MSC constructs in the context of testing. The second variety of tMSCs take the point of view of the system, containing an instance that represents the system and one instance for each PCO.

Finally, normal MSCs are accepted as test MSCs by our tool, in order to be able to take as input MSCs resulted from the simulation of an SDL model, for example. In such an MSC, the system is modeled by the instance(s) represented in the MSC, and the tester is represented by the outer border (environment) of the MSC.

A test MSC should be in exactly one of the forms described above. Messages towards the system (from a tester to a PCO in the *pvTester*, from a PCO to the system in a *pvSystem*, or from the ENV to the system in a *normal* MSC) are interpreted as TTCN sending events. Messages coming from the system are interpreted as TTCN receiving events. Therefore the exchange of messages represented in all the three MSCs in Figure 1 corresponds to the TTCN sequence:

```
p1!m1
      p2?m2
```

Note that one of the limits of *normal* MSCs is that they do not allow for the description of PCOs.

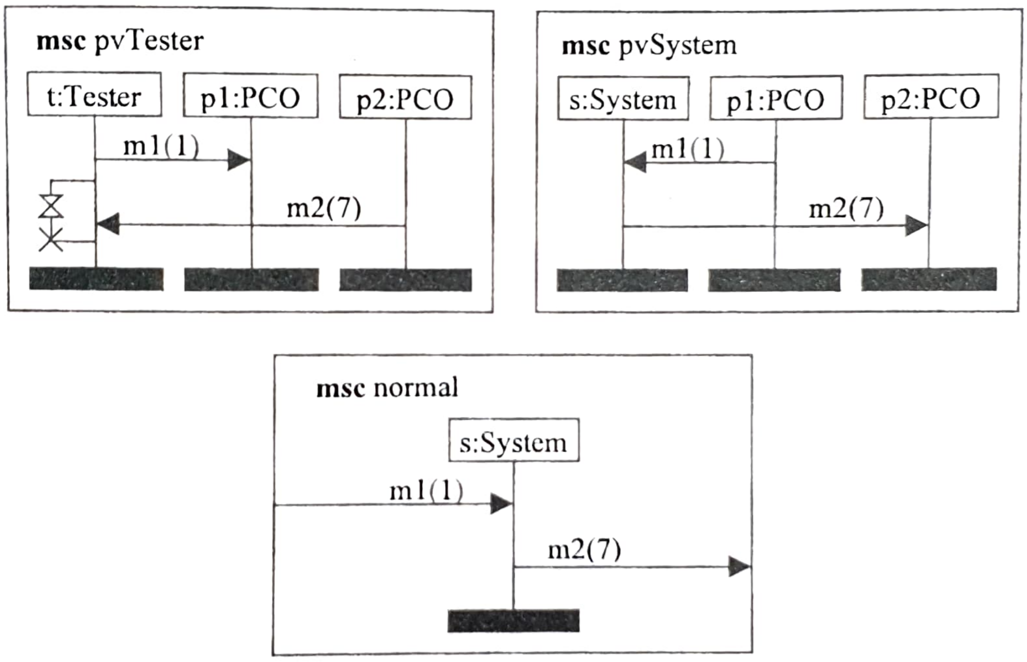


Figure 1. The three varieties of test MSCs

A message that should but cannot be sent produces a failure of the test. Similarly, a message that should be received but is not, or is received but with non-matching parameters produces a failure of the test.

As mentioned in [7], one inconvenient when using MSCs for describing test cases is that they lack constructs for expressing constraints on data (parameters of the messages). To alleviate this limitation, we used the fact that the MSC language gives no interpretation to the values of parameters of messages. We defined a set of generic specifiers that may be used in MSCs: *AnyValue*, *ValueList*, and *Range*. Still, the amount of constraints we are able to express using these specifiers in MSC is limited, due to the lack of variables. This is one reason for which TESTPLAYER uses a second language for describing tests, TDL (see Section 3).

Besides messages, three additional categories of constructs have a meaning on a Basic test MSC, but only when they appear on a *tester* instance: *stop*, *test timers* and *coregions*.

A *stop* construct on a *tester* bar means that the test stops immediately with a PASS verdict.

A *timer set* describes the action of the tester setting a timer. If the *set* is followed by a *reset* and the tester does not reset the timer *before it is triggered* (but after executing all the other constructs that appear before the *reset*) – test failure is assumed. Otherwise, the timer is reset (deactivated) and the test continues.

If the *set* is followed by a *timeout* and if the timeout occurs before the tester executes all the constructs that appear between *set* and *timeout* on the MSC, then test failure is assumed. Otherwise, after executing the constructs appearing before the *timeout*, the tester *waits* until the timer is triggered.

MSC timers can express a variety of timing constraints needed in a test case. The timing constraints employed in test cases usually take the following form: the time span between the event a and the event b must be between x and y time units. The way of expressing this with timers in shown in Figure 2.

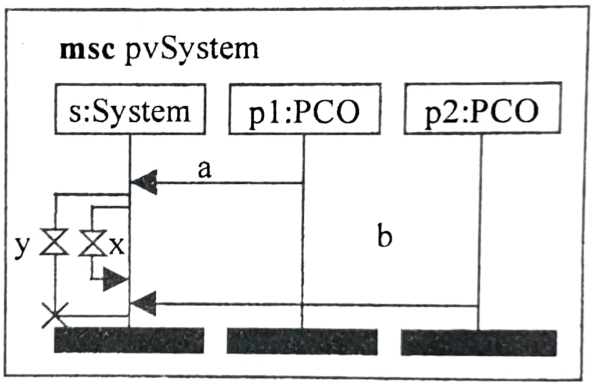


Figure 2. Delay intervals expressed with timers

A *coregion* on a *tester* bar means that the messages that come from the system (PCOs) within the boundaries of the coregion may come in any order. As a difference from Z.120, *only response messages* coming from the system and *timer timeouts* may be represented within a coregion. This is because the tester must be controllable, so we cannot allow message outputs and other actions for which we do not specify the precise order.

2.2 Hierarchical tMSC

MSC'96 introduces a set of structural constructs for organizing Basic MSCs into High-Level MSCs using sequential, alternative and repetitive composition, parallel composition and exception handling composition. For describing test cases one may use sequential, alternative and repetitive composition. Two MSCs which are to be composed *must* define the behavior of the same tester(s) on the same system (PCOs). The semantics of these composition operators is natural.

- The *sequential composition* of two tMSCs describes the tester that first executes the constructs specified by the first MSC and then, if it did not fail in the meantime, executes the constructs specified by the second MSC.
- The *repetition* of a tMSC describes the tester that repetitively executes the constructs specified by the MSC until the boundary of the repetition is reached or until the test fails at some point.
- The *alternative composition* of two tMSCs describes the tester that executes either the constructs specified by the first MSC or the constructs specified by the second MSC. The choice is made depending on the first construct from the two tMSCs, which *must* be a message from the system to the tester or a timer timeout (note that the timer timeout is possible only when the alternative is the follows by sequential composition another tMSC in which the timer is set). Depending on what event gets first to the tester, the tester chooses one alternative or fails (if neither one is eligible).

2.3 tMSC vs. TTCN

Let us recapitulate what we have said about the semantics of tMSCs, analyze their power of expression and see what are the strong and weak points with respect to TTCN. It is worth noting that not all the three flavors of test MSCs have the same power of expression. There are useful constructs mentioned above that are given an interpretation when they are on a tester bar and have no meaning on another bar: timers, coregions, actions. But only tMSCs that take the point of view of the tester (like *pvTester* in Figure 1) represent explicitly the testers as MSC instances. On tMSCs that take the *point of view of the system* or on *normal* tMSCs, we cannot put timers and that may be too strong a restriction for real-world test cases.

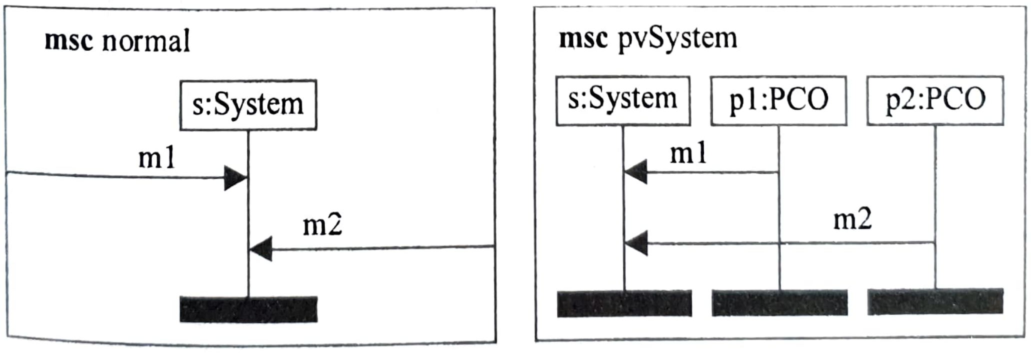


Figure 3. Visual order not specifying completely the tester

Moreover, considering the formal semantics of MSCs [4,11] the tMSCs that take the *point of view of the system* and *normal* tMSCs seem inappropriate to express the operational behavior of a tester. The normative semantics of MSC enforces a *partial order* between events (the sending and the receiving of a message are considered different events) called *visual order*. The visual order of the events in *pvSystem* and in *normal* in Figure 3 is:

$$\text{sending}(m1) < \text{receiving}(m1)$$
$$\text{receiving}(m1) < \text{receiving}(m2)$$

sending(m2)< receiving(m2)

The visual order does not say anything about the order between sending(m1) and sending(m2), which are precisely the two events that interest the tester: the tester must know in which order to send the two events. The conclusion is that among the three flavors of tMSC, the one that takes the *point of view of the tester* is the most appropriate for describing test cases. The other two kinds of tMSCs are necessary for compatibility with other tools (ex. an SDL simulator).

The constructs for which we have an interpretation in the context of testing, mentioned above, give MSC a power of expression comparable to that of the behavioral part of TTCN. MSC timers may express everything that is described in terms of timers in TTCN. MSC composition operators make it possible to express branching behavior in the same way as in complex TTCN trees.

When comparing tMSCs with TTCN, we must keep in mind that the goal of TESTPLAYER is to provide a lightweight solution for testing reactive systems. The one important advantage of using MSCs to express test cases is that MSC is a visual, intuitive and mature enough formalism. From this point of view TTCN is known as being cryptic and unreadable. This argument becomes important when coming to industrial test suites described on hundreds of TTCN tables.

Another argument in the favor of MSC is reusability. If a tool is able to interpret *normal* MSCs as test MSCs, (TESTPLAYER is one such tool) then MSCs constructed when designing a system may be used directly when testing it. The same happens with MSCs automatically generated from a simulation, for example.

The disadvantages of tMSCs with respect to TTCN were already mentioned and we reiterate them. There is no notion of test verdict in tMSCs. If the events occur as prescribed by the MSC, the test passes. If anything goes wrong, the test fails. There is no possibility to explicitly state a verdict in MSC except for the STOP statement. The STOP statement, not being parameterized, gives always the same verdict which in TESTPLAYER is PASS. INCONCLUSIVE verdicts cannot be expressed either implicitly or explicitly.

The data constraints that may be expressed in MSCs are far less powerful than those from TTCN. MSC is untyped, messages and parameters are not interpreted and are just strings of characters. Therefore, no type checking is possible in MSC. This point may be affected by the adoption of the future version MSC2000. It is desired that data aspects be formalized in MSC2000: signals and data types be defined before they are used, using ASN.1 or some other formalism.

MSC is a declarative language, while TTCN is rather imperative. Things that can be done in TTCN with variables, like remembering the value of a parameter of an incoming message and using this value later, cannot be done at all in MSC. This point may also be affected by the adoption of MSC2000, which defines some notion of variable.

We conclude that, although very appealing, MSC alone cannot be used to express real test cases. However, the strong points of MSC can be exploited if it is used in connection with a second layer of formalism for expressing tests, layer that will provide the missing bits. This kind of architecture seems ideal for a lightweight and open approach towards testing and we retained it for TESTPLAYER.

3. The TDL

The second layer on which tests may be represented in TESTPLAYER is the Test Description Language (TDL). There is no test execution engine in TESTPLAYER that takes tMSCs and executes them directly. Instead a tMSC is translated in a TDL script. At this level the programmer may intervene and provide the parts of her test case that she could not express in MSC (explicit verdicts, use of variables, etc.). The execution engine in TESTPLAYER executes TDL scripts.

The Test Description Language is in fact a set of primitives, with a precise semantics and an implementation that may be embedded in any language that allows for extensions (C, TCL, Python, etc.). The primitives correspond to TTCN executable instructions and to certain MSC constructs. They differ from MSC in that they are imperative rather then declarative. They provide the missing bits, notably variables and verdicts, and are also an implementation model for tMSCs.

3.1 The execution model

The execution model is designed such that the execution engine may execute a single threaded test at a time. The execution engine sees the test as a sequence of calls to the test primitives. These calls must be sequenced and may not be launched in parallel.

Time is a discrete numeric value in TDL, specific to a tester, which gives the number of milliseconds elapsed since the tester has begun to execute. The Now primitive gives access to the current time.

TDL has primitives for sending signals to the system, expecting signals to come from the system and manipulating timers. A tester may exchange signals with the system under test on a per-PCO basis: it sends signals through PCOs or it expects signals to arrive on PCOs. PCOs are named and must be open before they are used and closed after they are no longer necessary. This is because for each newly open PCO, the engine establishes a communication connection with the system under test and allocates a queue for keeping the incoming signals.

The *outgoing signals* are sent immediately to the system through the communication connection corresponding to the specified PCO. A background task receives the *incoming signals*, stamps them with the precise moment when they arrived and then put them in the queue of the corresponding PCO. Thus the primitives for expecting signals access the signals from the queue and have also the possibility to know the exact moment at which the signal arrived. This is useful for expressing different timing constraints. A signaling convention is put in place so that the task executing the primitives and the background task retrieving messages may synchronize; thus a primitive for expecting signals may wait until a certain signal enters the queue.

In our model, timers may be set to a certain period of time and reset. The tester may wait for a timer or simply check if it has expired. A timer that expires is not put into a queue (like in SDL): if there is a primitive waiting for it then the primitive is signaled so it may continue its execution.

The only additional testing primitive that does not refer to PCOs, signals or timers is the one called Verdict. This primitive causes the end of a test and the de-allocation of all resources used for the ongoing test (queues, semaphores, OS timers, etc.).

3.2 The TDL primitives

Open/Close are used to open and close connections between the tester and a PCO of the system.

Syntax[1]:	Open [host] [pco] Close [pco]
Description:	Before sending or receiving a message through a PCO, a tester must open the connection to that PCO. This allocates the resources necessary to handle the new PCO. When the PCO is no longer used, the tester should close it. This de-allocates the resources used to handle the PCO.
Verdicts:	Open fails if a connection with the system under test cannot be established or if the specifies PCO name is not supported. Close fails when the specified PCO name is not the name of an open PCO.
Example:	Open rtos1 ChannelA Close ChannelA

Expect is used to specify that the tester expects the system to send a certain signal.

Syntax:	Expect <signal> [[with] <param list>] [via <pco>] [from <src-pid>] [time "[{store var}] [{after time}] [{before time}]"]
Description:	**Expect** is used to specify that the tester expects the system to send a response signal on a certain PCO. As in TTCN, the user must specify the expected signal and its parameters. Different constructors are available for specifying generic paramcters: lists of possible values, intervals of possible values, *AnyValue*. If the tester communicates with different PCOs, the name of the PCO on which the signal is expected *must* be specified. An optional source PId must be specified, its interpretation depending on the system (for SDL systems the PId may represent the PId of an SDL process). To allow a more fine grained representation of timing constraints, in TDL one may express timing constraints not only through timers. In the case of Expect, one may request that the time of arrival of the matched message (the moment when it was put in the queue) be after a certain moment, before another moment, or simply stored in a variable for future reference. The semantics of the Expect primitive is such that if the specified message name and parameters match the message that is in the head of the queue of the specified PCO, the timing constraints are met and the source PId is matched, then the primitive completes successfully and pops out the message from the queue. If one of these conditions are not met, Expect gives a FAIL verdict. If at the moment Expect is executed there is no signal in the PCO queue, the primitive waits until a signal arrives or until the moment specified in the before clause passes.

[1] The syntax presented here is the syntax used for calling the TDL primitives from the TCL language [12].

Verdicts:	Expect fails if the expected signal does not match the signal that is in the head of the queue of the specified PCO (because of the signal name, parameters, source PId or timing constraints). Expect fails also if the queue is empty and no signal arrives in due time.
Example:	Expect CACK with {interval 0..7} via ChannelA time {before [Now + 1000] }

Output is used to send signals from the tester to the system.

Syntax:	Output <signal> [[with] <param list>] [via <pco>] [to <dest-pid>] [time "[{store var}] [{after time}] [{before time}]"
Description:	Output sends the specified signal to the system on the specified PCO. The parameters of the signal must have concrete values i.e. no wildcards are allowed. The PCO to which the signal goes must be specified if the tester communicates with more than one PCO. An optional destination PId may be specified, its interpretation depending on the system (for SDL systems it may be the PId of an SDL process). Time constraints may also be specified and refer to the moment when the message is sent.
Verdict:	Output fails if the message cannot be sent to the system (message undefined, bad parameters, network failure, timing constraints not met, etc.).
Example:	Output CC with 0x01001 via ChannelA time {before $t1 + 10 }

TimerSet, **TimerReset**, **TimerWait**, **TimerQuerySignaled** are the TDL primitives for manipulating TTCN-like timers.

Syntax:
```
TimerSet <name> <duration>
TimerReset <name>
TimerWait <name>
TimerQuerySignaled <name>
```

Description: TimerSet sets a named timer to trigger after a specified period of time. TimerReset resets a timer to the inactive state. TimerWait causes the calling task to wait util the timer is triggered. TimerQuerySignaled makes it possible to query the timer in order to see if it has triggered or not.

Additionally timers may be used wherever an "expected" message may be used. For example, in the Expect primitive, we can write «Expect timer t » which is equivalent with «TimerWait t ». Such a use of timers becomes useful in primitives like Coregion or Alternative, described below.

Verdict: Timer primitives never provoke the failure of a test.

Example:
```
TimerSet t 1000
TimerQuerySignaled t        # returns FALSE
TimerWait t                 # waits for 1 sec. minus the delay since the TimerSet
TimerQuerySignaled t        # returns TRUE
TimerReset t                # resets the timer
TimerQuerySignaled t        # returns FALSE
```

Alternative is the TDL primitive allowing to describe branching behavior based on what message between a given set is received.

Syntax:	Alternative ({ <signal> [[with] <param list>] [via <pco>] [from <src-pid>] [time {[store <var_name>] [after <time>] [before <time>]}] })+
Description:	Alternative resembles Expect, except that multiple messages (and timers) may be specified. If one of the specified messages is in the head of the corresponding PCO queue or one of the specified timers is triggered, then the primitive behaves as Expect, terminating successfully and returning the index of the matched alternative. If none of them occurs, the engine waits until no one can occur any longer (i.e. all the PCOs involved have some message in the queue, but is not an expected one) and after that it raises a FAIL verdict.

Verdicts:	Alternative fails when none of the specified signals and timers has occurred and no one can occur any longer.
Example:	Alternative "CACK with 1 via ChannelA" "DC with 1 via ChannelB" "timer t"

Coregion is the TDL primitive for describing a set of messages and timer timeouts that come in an unspecified order.

Syntax:	Coregion ({ <signal> [[with] <param list>] [via <pco>] [from <src-pid>] [time {[store <var_name>] [after <time>] [before <time>]}] })+
Description:	Coregion resembles Alternative, except that it waits for all the events to happen, and not just for one of them. It has the same meaning as a coregion in tMSC.
Verdicts:	Coregion fails when an unexpected signal arrives on a PCO (or is already in the head of the PCO queue) and this unexpected signal impedes another expected signal to get in front of the queue.
Example:	Coregion "CACK with 1 via ChannelA" "CC with 1 via ChannelB" "timer t"

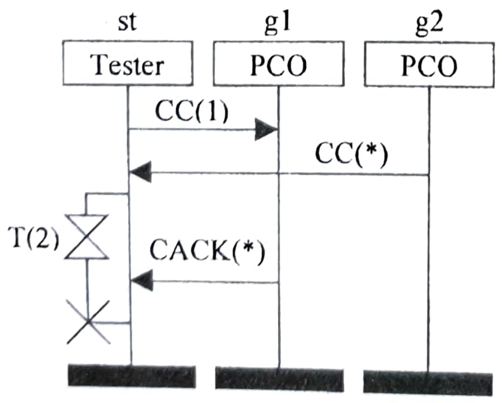
Verdict is the TDL primitive for explicitly specifying a verdict for the ongoing test case.

Syntax:	Verdict PASS | FAIL | INCONCLUSIVE
Description:	Verdict terminates the ongoing test, recording the test verdict in the test log and de-allocating all the used resources.
Verdicts:	Whatever verdict is specified in the parameter
Example:	Verdict INCONCLUSIVE

3.3 Translating tMSCs into TDL scripts

In what follows, we will use the name TDL both for denoting the "abstract" primitives presented above and the TCL [12] language extended with the TDL primitives. TDL script always stands for extended TCL script.

The translation of tMSC into TDL scripts is almost straightforward: for each tester described in a tMSC, a script is generated. Usually a tMSC describes only one tester, but in the case of *tester point of view* tMSCs (see Figure 1) there may be more. Tester outputs are translated into Output statements, tester inputs into Expect statements, timers into timer statements, coregions into Coregion statements and so on. An example of translation for a Basic MSC is given in Figure 4.



```
Open g1; Open g2
Output CC with 1 via g1
Expect CC with star via g2
TimerSet  t  2
Expect CACK with star via g1
if { TimerQuerySignaled t } {
        Verdict FAIL
}else {
        TimerRest t
}
```

Figure 4. tMSC translated in TDL

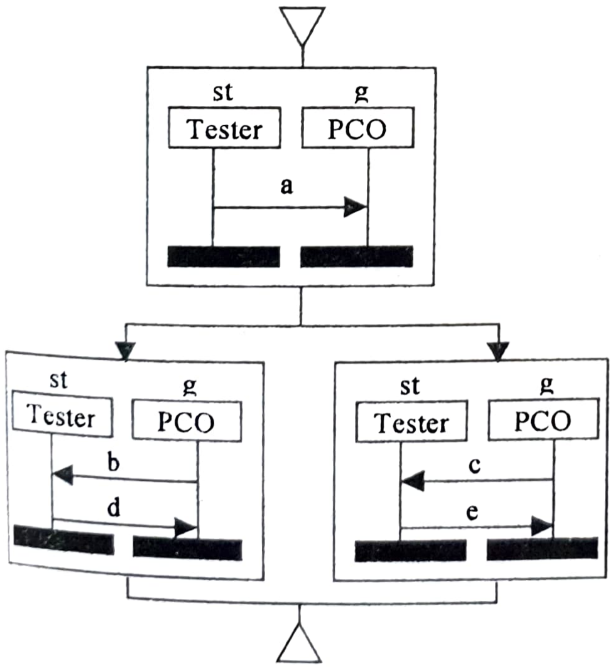
For High-level MSCs, the sequential composition of MSCs causes the concatenation of the scripts described by the operand MSCs. The alternative composition of MSCs is translated using the Alternative statement and repetitive composition is translated using the repetitive construct from TCL (or from any other language that may host the TDL primitives). An example is given in Figure 5.

The TDL primitives combined with an imperative programming language providing alternative and repetitive constructs, variables and all that it usually available in such a language, provide a power of expression that covers all that can be expressed in the behavior description of a TTCN test case. TDL primitives are implemented as an extension to the TCL language [12] but they might as well be integrated in another language like C, C++, Python or Java.

Using a visual formalism like MSC and a small set of primitives integrated in a scripting language (TDL) ensures a small learning curve. TDL scripts being directly interpreted by the TESTPLAYER execution engine, usual programming techniques like step-by-step execution and the use of breakpoints may be applied to the execution of a test.

4. TESTPLAYER Open Architecture

The goal in designing TESTPLAYER was to provide an open architecture for testing, based on the lightweight formalisms of tMSC and TDL, and which would provide customizability and power of expression sufficient to be applicable to a large number of real systems and test cases.



```
Open g
Output a via g
set r1 [Alternative "b via g" "c via g"]
if { $r1 == 0 } {
            Output d via g
} elif { $r1 == 1 } {
            Output e via g
}
```

Figure 5. High-level MSC translated in TDL

In our architecture (see Figure 6) we distinguish between two phases of applying a test: the preparation phase and the execution phase. For preparing a test, the user may either provide the tMSC description of the test case which will be translated automatically in TDL, or she may provide the TDL script directly. The MSC-to-TDL compiler (*TDL Gen* on Figure 6) does the translation from the tMSC to TDL following the rules depicted in the previous section.

Also in the preparation phase, the user must provide the description of the signals and data types exchanged between the tester and the system. This information is needed at execution time (when it is captured by component *(1)*, described below). The user may supply the signal and type information either by providing directly the body of the component *(1)* or – more likely – by providing the information in a formalism like SDL or ASN.1 from which the component (1) is generated automatically by a sort of compiler (*Stub Gen* on Figure 6).

At execution time, the core of the TESTPLAYER environment is the *TDL execution engine* – the implementation of the TDL primitives. Our paradigm of executing tests being based on it, this is a fixed part of the architecture i.e. it cannot be customized. The implementation of the test primitives in the *TDL engine* is a static library of functions written in C. However, it may be used from a variety of host languages such as C, TCL, Pascal or any other language that allows for the use of static libraries. For each host language (except C which is the native language in which the TDL engine is written) there must be a wrapper that makes the primitives accessible from the language.

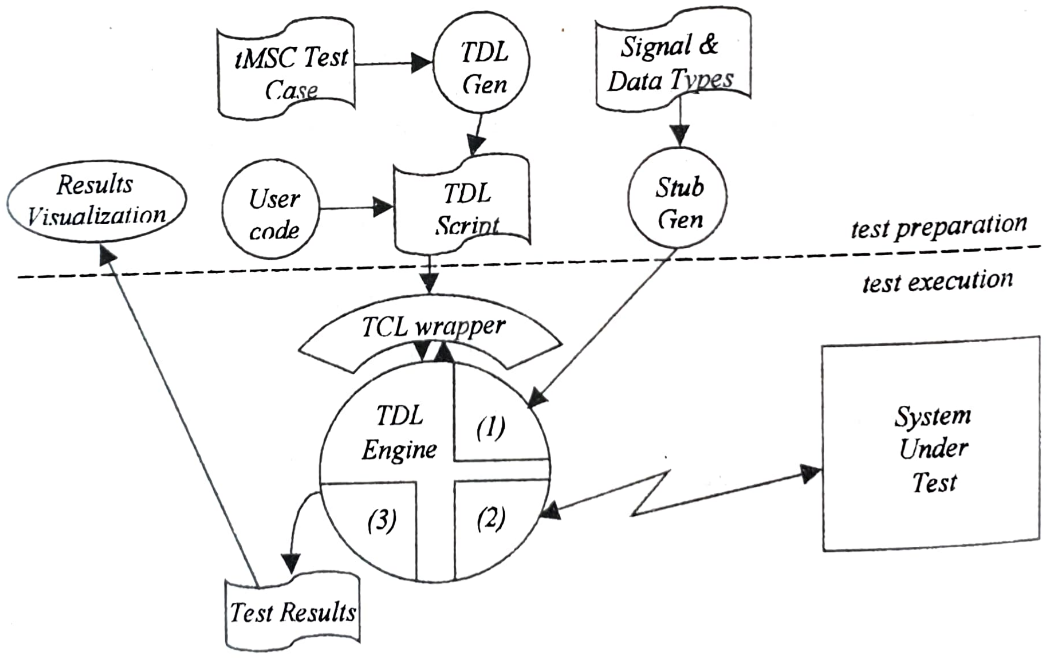


Figure 6. TESTPLAYER Architecture

The most common case of usage envisioned in TESTPLAYER is from TCL. For that purpose, a *TCL wrapper* library was written, which makes available the test primitives as TCL commands. We used this set of commands for exemplification in the previous section and we called TDL the enriched TCL language.

The *TDL engine* uses a set of utility components, which may be customized by the user. At implementation level, such a component would be a static library implementing a certain interface. The user is free to provide any replacement for a component, with the only restriction that it implements the *same* interface.

The first customizable component is *Signal Encoding/Decoding*, denoted by *(1)* on Figure 6. This component contains callback functions for signal and data encoding/decoding/matching. The tester communicates with the system under test using a specific protocol, as we will see later, and the functions contained in component *(1)* are called to pack/unpack the signals and data to be transmitted/received. Component *(1)* also contains the functions that match an incoming message towards a generic message specification given in an Expect, Alternative or Coregion command.

As mentioned above, the *Signal Encoding/Decoding* component may either be provided as such by the designer of a system, or it may be generated from the SDL or ASN.1 description of the system signals and data types.

The *Protocol* component denoted by *(2)* on Figure 6 implements the *protocol* used by the tester to communicate with the system under test. TESTPLAYER may be used to apply tests in different configurations, in which the tester and the SUT may be on the same machine or on two different machines connected by a network, a serial line, etc. To achieve such a generic architecture, the communication functionality must be kept in a separate component in order to allow for changes in the underlying communication primitives. The *Protocol* component *(2)* must implement a simple interface that basically knows to open and close connections, raw send binary data to a connection and do blocking reads on a connection. Beyond this, the user is free to use whatever suitable means to implement the communication between the tester task and the SUT (TCP/IP, some proprietary protocol, OS IPC, etc.)

The *Test Reports* component *(3)* formats and stores the partial or final results of a test in a log. We kept this functionality in a separate component because a user may require that the results of her tests be stored in a specific format: a database, a log file with a proprietary format, etc. Any replacement for this component must implement a simple interface containing the following functionality: opening/closing a log, writing a test event to a log, writing a test verdict to a log.

After the execution of a test case or suite, the results may be visualized and one may find the initial MSC construct corresponding to a test event that appears in the log, using the *Results Visualization* tool. This tool accesses the information contained by the test log, using a component for reading the log similar to *(3)*.

Thus we have obtained a testing architecture based on the interpretation of test MSCs and on the Test Description Language, but in which the parts describing system signals and data types, the communication protocol between the tester and the system, and the formatting of test results are left open and customizable.

5. TESTPLAYER for *Object*GEODE

TESTPLAYER for Object*GEODE* is the complete TESTPLAYER solution for systems modeled in SDL [2] with ObjectGEODE [13] and for which code is automatically generated with the ObjectGEODE SDL-C Code Generator. It is an instantiation of all the generic components and tools described in the previous section. The concrete architecture after the instantiation is outlined in Figure 7 (the instantiated tools and components are drawn on gray background).

First of all, a real Stub Gen (see Figure 6 and Figure 7) is provided, a compiler that automatically generates the code for the component (1) from the SDL description of the system. This compiler is written using the SDL API which is part of the ObjectGEODE toolbox. The generated component (1) contains the encoding/decoding functions for the signals and data types defined in the SDL system.

(2) is the component in charge with the communication between the tester and the system. The instantiation means choosing an underlying protocol, which in our case was TCP/IP. The two parties involved in communication, the tester and the system, must understand each other, so an adapter had to be added on the system side. To describe this adapter task, we have to depict shortly what the SDL-C Code Generator generates.

A system generated by the ObjectGEODE SDL-C Code Generator is a set of communicating tasks. A task may physically be a process or a thread, depending on the target operating system. A task may correspond to an SDL process instance, to a process instance set, to a block or to the entire system, depending on the configuration parameters given to the generator.

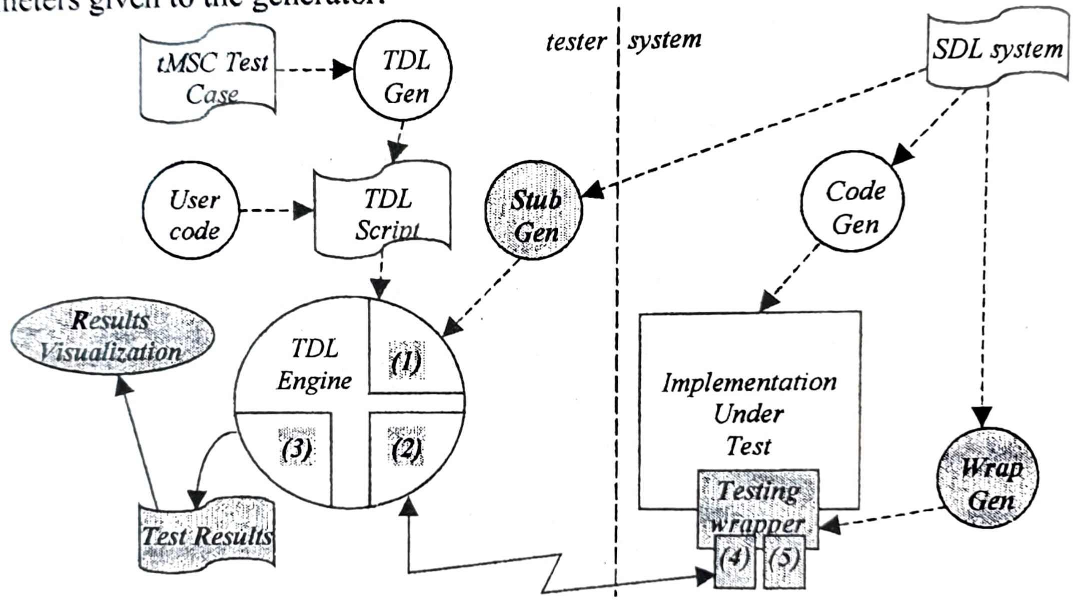


Figure 7. TESTPLAYER Concrete Architecture for ObjectGEODE

We are interested in the communication between the system and the environment. For a generated system, the environment is a task as any other task. Messages emitted to the environment are put in the queue of that task, which in turn may emit messages towards the tasks of the system. The environment task may (and generally will) be described by the designer of the system. In case it is not described, a default replacement of the task is generated, but this replacement does nothing else than deleting all the messages that arrive in its queue.

For testing purposes, we replace the environment task with a task that communicates with the tester (using our protocol implemented by component *(2)*). The external communication task is represented on Figure 7 under the name *testing wrapper*. This task exposes an interface to the system organized as a set of PCOs [2].

[2] PCOs do not exist in the SDL system, so we have to map them to something that exists. We consider that a PCO may correspond either to a channel between the system and the

Concretely, this means that the external task behaves as a TCP server and accepts at most one TCP connection for each exposed PCO. A tester that wants to communicate with the system on a specific PCO must call the Open primitive, which will open a TCP connection to the server and will announce the name of the PCO on which it acts. After the TCP connection corresponding to a PCO is established, every signal coming to the server on that TCP connection will be decoded and routed to the corresponding destination task without any modification and every signal coming from the system on the PCO will be encoded and forwarded through the corresponding TCP connection.

The *testing wrapper* contains two sub-components: *(4)* which implements the communication protocol (very similar to *(2)* on the tester side) and *(5)* which implements signal and data types encoding/decoding (similar to *(1)* on the tester side). An automatic generation tool that generates the task from the SDL specification of the system is provided.

The results formatting and storage component *(3)* is also instantiated for ObjectGEODE. It stores the test events and verdicts in an ASCII format in the *test results* repository. A visualization tool can read the repository and make the link back to the MSC source of a test case.

The conclusion drawn from applying TESTPLAYER to ObjectGEODE systems is that the instantiation of the open parts of the TESTPLAYER architecture is not difficult to achieve, and that the resulted toolbox may be successfully used on large SDL systems.

5.1 An example

We prototyped the architecture described in the previous section in order to validate it on real examples. Our prototype provides all the necessary tools and components for testing a class of systems, namely systems designed and generated with ObjectGEODE on WindowsNT. This section shows how we can test a system with the prototype TESTPLAYER.

We consider an SDL system describing a Cards Game Server. The high-level architecture of the system is shown in Figure 8 (for brevity, not all the signals are shown).

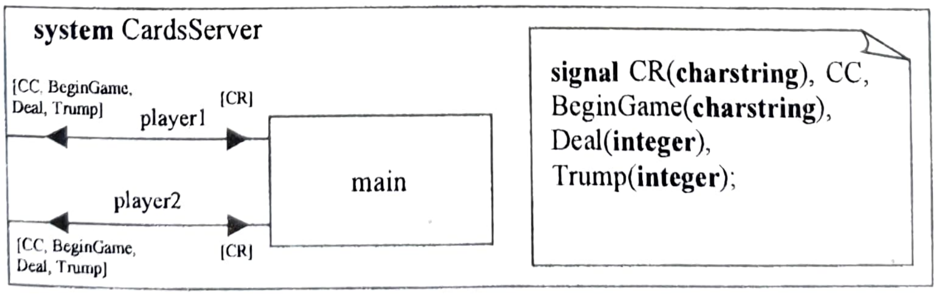

system CardsServer
[CC, BeginGame, Deal, Trump]
[CR]
player1
main
player2
[CC, BeginGame, Deal, Trump]
[CR]
signal CR(**charstring**), CC, BeginGame(**charstring**), Deal(**integer**), Trump(**integer**);

Figure 8. SDL system for a card game server

environment, or to a set of such channels, or to all the channels when the system has only one PCO, and place is left for other mappings. One such mapping says what means that a signal is coming from the system through a certain PCO and, also, what is the destination task corresponding to a signal sent to the system on a PCO.

We would like to test the connection phase of the game. The connection involves both players that may connect to a game, so the tester will have to take the place of both of them. Each channel linking the system and the environment will be therefore considered as a PCO. For testing the connection phase we will re-use an MSC that we already have from the requirements specification of the system (Figure 9)

After providing all the configuration parameters needed by the ObjectGEODE SDL-C Code Generator in the `CardsServer.cfg` file, we generate the executable system and the testing wrapper task in one pass, by invoking:

```
st_gen_sys CardsServer
```

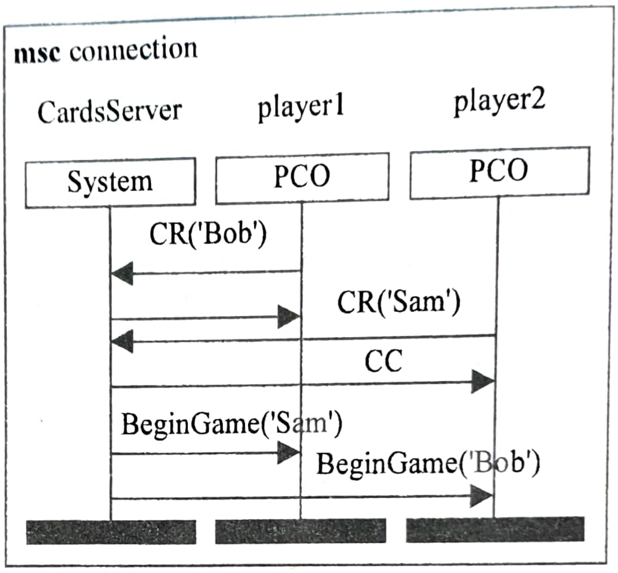
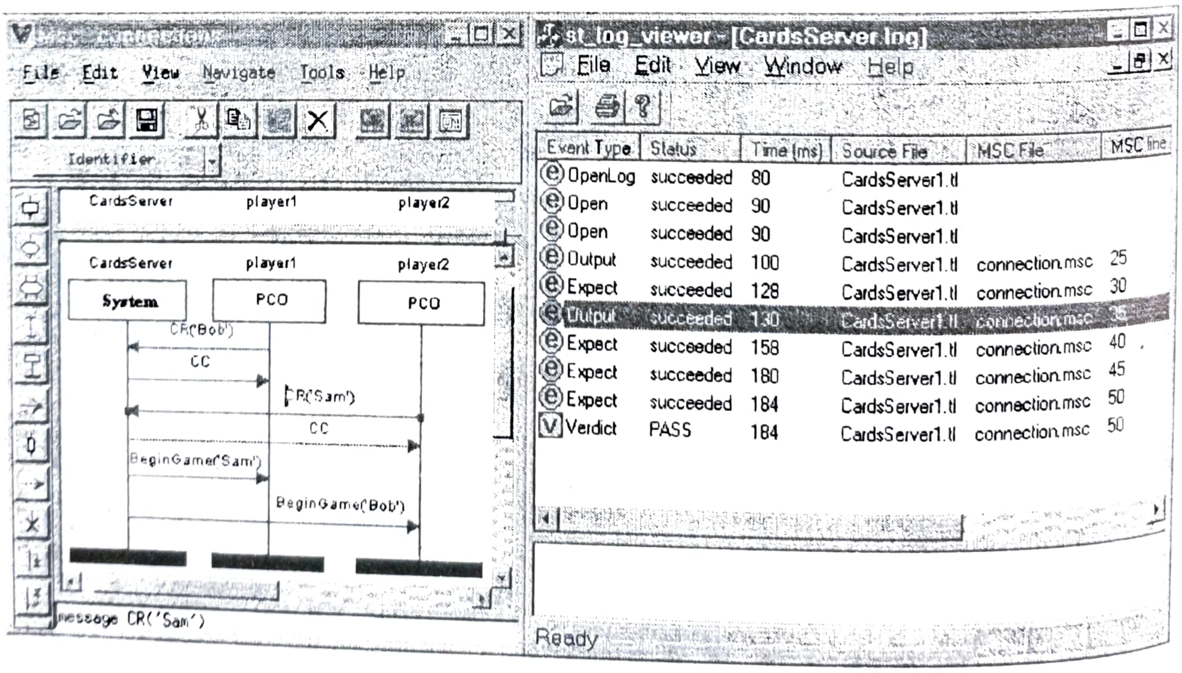


Figure 9. Connection phase of the Cards Game



Event Type	Status	Time (ms)	Source File	MSC File	MSC line
OpenLog	succeeded	80	CardsServer1.tl		
Open	succeeded	90	CardsServer1.tl		
Open	succeeded	90	CardsServer1.tl		
Output	succeeded	100	CardsServer1.tl	connection.msc	25
Expect	succeeded	128	CardsServer1.tl	connection.msc	30
Output	succeeded	130	CardsServer1.tl	connection.msc	35
Expect	succeeded	158	CardsServer1.tl	connection.msc	40
Expect	succeeded	180	CardsServer1.tl	connection.msc	45
Expect	succeeded	184	CardsServer1.tl	connection.msc	50
Verdict	PASS	184	CardsServer1.tl	connection.msc	50

Figure 10. The Results Visualization Tool

By default this will generate the executable system (`CardsServer.exe`) containing also the testing wrapper, generated in such a way that each channel linking the system and the environment becomes a PCO. This configuration suits our purpose.

Other configurations are possible for other purposes, like: one PCO for the entire system (grouping all the channels) or n PCOs – each one for a specified group of channels.

We now generate the extended TCL interpreter customized for use with our CardsServer system (i.e. containing the signal and type information specific to our system). The starting point is the SDL specification again:

```
st_gen_interp CardsServer
```

The executable interpreter (called `tdlsh`) that is generated corresponds to the *TDL engine* plus the *components (1), (2)* and *(3)* in Figure 7. The type information contained in the SDL system is used for generating component *(1)*. The other components are not rebuilt, they are taken from a library and linked together.

We have two possibilities for using the executable system and the TDL interpreter: either we generate the TDL script from the MSC(s) and execute it in batch mode, or we try an interactive session in which we feed whatever TDL commands we want to the command prompt of the TDL interpreter.

The first alternative involves first of all generating the TDL script from the MSC description of the test case. We can do this with:

```
st_gen_tests connection.msc
```

and we obtain the TDL script called `CardsServer1.tdl` containing the following code:

```
OpenLog CardsServer
Open player1
Open player2
Output CR with Bob via player1
Expect CC via player1
Output CR with Sam via player2
Expect CC via player2
Expect BeginGame with Sam via player1
Expect BeginGame with Bob via player2
Close player1
Close player2
Verdict:PASS
```

To run this test, we must first start the system (`CardsServer.exe`) in a console, and then invoke:

```
st_exec_tests CardsServer1.tdl
```

in another console. This command will run the test in batch mode and will write the test events and verdict in `CardsServer.log`.

This log may be visualized with `st_view_log` and the MSC construct corresponding to each event appearing in the log may be highlighted automatically in the ObjectGEODE MSC editor (see Figure 10). For FAIL verdicts, the failure reason is also stored in the log.

For using the TDL interpreter in interactive mode, one must start the system in a console and the `tdlsh` in another console. Then, at the `tdlsh` prompt he may type commands like the ones generated from the MSC. The outputs and verdicts of the commands will appear on the console.

6. Conclusions

We have presented an original framework for expressing and executing conformance test cases. Our architecture is based on two formalisms: MSC, which is a standard language for expressing execution traces in terms of message exchanges, and TDL which is our own language and set of primitives for expressing actions of a tester. We have shown that the combined use of the two formalisms provides a high power of expression while preserving the original simplicity and ease of the graphical MSCs.

Our ideas are implemented in a toolbox which can be used for executing tests on systems designed and generated using ObjectGEODE [13]. The architecture of the toolbox is such that it can be rapidly customized to work on different platforms (operating systems, network protocols, etc.) and different target systems. An industrialization of this tool suite by Verilog is planned.

References

1. ATTOL Testware, *ATTOL UniTest V3.3 Technical White Paper*, 1999
2. ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*, 1996
3. ITU-T Recommendation Z.120. *Message Sequence Charts*, 1996
4. ITU-T Recommendation Z.120. Annex B. *Formal Semantics of Message Sequence Charts*, 1996
5. ISO/IEC International Standard 9646-3. *OSI-Open Systems Interconnection, Information Technology – Open Systems Interconnection Conformance Testing Methodology and Framework. Part 3: The Tree and Tabular Combined Notation (TTCN)*, 1992.
6. ISO/IEC International Standard 9646-1/2/3. *OSI-Open Systems Interconnection, Information Technology – Open Systems Interconnection Conformance Testing Methodology and Framework*, 1992.
7. J. Grabowski, T. Walter. Visualization of TTCN test cases by MSCs. In *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC, Informatik Bericht Nr. 104.* Humboldt Universität Berlin, 1998.
8. J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts.* Ph.D. Thesis, Universität Bern, 1994.
9. J. Grabowski, D. Hogrefe, R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In: *SDL'93 – Using Objects.* North-Holland, 1993.
10. J. Grabowski, D. Hogrefe, I. Nussbaumer, A. Spichiger. Test Case Specification Based on MSCs and ASN.1. In: *SDL'95 – Proceedings of the 7th SDL Forum,* Sept. 1995, Oslo, Norway. North-Holland, 1995
11. S. Mauw, M.A. Reniers, An Algebraic Semantics of Basic Message Sequence Charts. In: *The Computer Journal,* 36(5), 1993
12. J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994
13. VERILOG. *ObjectGEODE Reference Manuals*, 1996

INP Toulouse, ENSEEIHT, 2 rue Camichel, 31000 Toulouse
Iulian.Ober@enseeiht.fr