Automated test generation from SDL/UML specifications

Alain Kerbrat[a], Iulian Ober[b*]

[a]Verilog, 150 rue Nicolas Vauquelin, BP1310, 31106 Toulouse cedex, France
Tel: (33) 5 61 19 29 39, Fax: (33) 5 61 40 84 52
e-mail: Alain.Kerbrat@verilog.fr www: http://www.verilogusa.com

[b]INPT-ENSEEIHT, 2 rue Camichel, 31000 Toulouse cedex, France
e-mail: Iulian.Ober@enseeiht.fr

Automated test generation from formal specifications presents a lot of promises, either in cost control or test suite correctness. The telecommunications domain is a field where some interesting tools begin to emerge, either prototypes or industrial strength tools. We present TESTCOMPOSER, a new industrial test generation tool. [2] TESTCOMPOSER allows the generation of conformance tests, from SDL/UML specifications. It comes with TESTPLAYER, which allows to execute test cases written with the MSC language. Both tools are part of *Object*GEODE, an industrial toolset for real time systems development.

1. Introduction

Test generation is an area on which many companies are focusing in search of productivity and quality gains. Current industrial practices include at best semi automated test generation techniques, with all the cost and maintenance problems of an error prone process. This process should be fully and automatically re-executable, with a complete understanding of the relationships between specification and tests, their meaning and their execution results.

Automating the test generation requires a precise and tractable description of the system to test. This is where using Formal Description Techniques (FDT) such as SDL brings most of their benefits. SDL (Specification and Description Language) is an international standard [16], defined over the years by the International Telecommunications Union as a common language for the specification of protocol norms. From 1988, a formal definition of the language semantics was added. These formally defined semantics allow to design reliable tools for the simulation, code generation and test generation.

In the telecommunications field, the FDT based test generation has been developed first by giving a concrete framework to conformance testing [1]. Conformance testing is a necessary step to ensure the compliance of various implementations of a given protocol to its standards. Then this approach has been formalized further in order to facilitate the emergence of tools [18]. However, the interest into FDT based test generation tools has been mostly academic up to the middle of the 90s.

*Iulian Ober's work is financed by a research grant provided by the CS (Communications and Systems) group
[2]This work has been initiated and funded by France-Telecom

TestComposer: automated test generation for real time critical systems

The main focus of this paper is TestComposer. TestComposer [11] is a test generation tool, which allows to generate automatically test suites from Sdl/Uml specification. While designed primarily for conformance black-box tests generation, its design principles make it applicable for grey or white box tests.

TestPlayer: a flexible approach for rapid test prototyping

TestPlayer [12] is a flexible toolbox for specifying, executing and getting reports for tests written using the graphical language Msc. The actual version of TestPlayer works for implementation coming from Sdl/Uml specification. It is however designed in a modular and open way, allowing the user to integrate its own extensions to perform tests in his own environment. Although built as a separate tool, TestPlayer can use TestComposer outputs, the two tools forming a complete test environment, from the test specification down to the test execution and reporting.

Structure of the paper

In section 2, we present the main features of the *Object*Geode [19] toolset.

Section 3 is devoted to the architecture and working principles of the tool TestComposer. We describe the different phases of the test generation process (test purposes edition and generation, test cases generation and test suite structuring). Then we give indications on the added value of this tool.

In section 4, we give an overview of TestPlayer. We show how it integrates with *Object*Geode and TestComposer.

2. *Object*Geode

*Object*Geode is a development toolset for real time and safety critical systems. It supports the joint use of Sdl, Msc and Uml.

Sdl is the Specification and Description Language [16]. Sdl is a formally defined language which allows to describe the architecture and the behavior of a real time distributed system. Originally defined in 1984 it was given formal semantics in 1988. In 1992, object oriented concepts were introduced, especially at the architecture and behavior level. 1996 was mainly for clarifications and strengthening of concepts.

Msc is the Message Sequence Charts language [17]. It is used to describe runs of the system, with a more or less abstract view of its architecture. Msc is especially useful for the description of nominal and error handling behaviors

Uml is the Unified Modeling Language. In the *Object*Geode's process, Uml is used for the analysis phase and for the data description, through the Class Diagram, Use Cases and Statecharts.

The combination of Sdl and Msc can be viewed as a formal extension of Uml, dedicated to hard real-time and safety-critical systems [7].

*Object*GEODE provides tools for the edition of these three languages, simulation, formal verification and C code generation. TESTCOMPOSER relies heavily on *Object*GEODE's simulator, so we give some more details on this tool.

The simulator tool

Validation and verification with *Object*GEODE is done with the Simulator tool. The Simulator allows to simulate the execution of the program, without having to actually execute it in a real environment. It can be seen as a sort of abstract debugger, as it allows to simulate the description step by step, to undo execution steps, to set break points and to watch the contents of variables and queues. Finally, it also allows to record, visualize as MSCs or replay some simulation sequences.

It is also more than a debugger, as it allows to perform automatic simulation, either randomly or exhaustively, with systematic comparison of the behavior with special state machines called observers. The simulator working principle is based on the model checking principle. Model checking [5, 15] consists in building a model of the system under analysis and to check the desired requirements on this model. For the simulator, the model consists in representing the state graph of the specification. Each state of this graph correspond to a different valuation of the system's variables. Each transition correspond to actions such as inputs, outputs, assignments, which can change the content of variables. The check itself amounts to a partial or complete exploration of the model.

The main advantages of this technique are that it can be automated and it is fast. Furthermore, model checking allows the easy production of counterexamples, when a property happens to be false.

The bottleneck of these techniques is the size of the state space to explore, which is exponential in the number of parallel processes of the system. In fact, for SDL-UML specifications, the size of the state space is usually infinite, as SDL allows unbounded process creation, and communication channels are based on unbounded queues. However,the simulator exploration engine works "on the fly", meaning that it builds this state graph on demand, according to the transitions it needs to explore. This "lazy" technique allows to check properties even on infinite state graphs. It will not allow to prove these properties, as proofs need exhaustiveness, but it is very useful for finding errors.

3. Production of test suites with TESTCOMPOSER

3.1. TESTCOMPOSER basics

TESTCOMPOSER working principles come from the conformance testing framework, as defined in [2] and formalized in [18]. Conformance testing can be viewed as a form of black box testing. The interactions between the tester and the System Under Test (SUT) are messages, exchanged through Points of Control and Observation (PCO).

TESTCOMPOSER is the result of the industrial transfer of two complementary research tools, TVEDA [14, 6] and TGV [9, 4]. TVEDA comes from the research laboratory France-Telecom-CNET, it provides for test purposes and test case generation, based on state space exploration combined with heuristics. TGV has been designed in the French research laboratories IRISA and VERIMAG, it provides test case generation based on efficient, on-the-fly state space exploration techniques.

The two prototype tools TVEDA and TGV offer complementary aspects of the test generation process. TVEDA's strengths are the analysis techniques of the formal specification. These techniques allow to derive from the specification meaningful behaviors for the test, which are referred below as test purposes. On the other side, TGV applies powerful formal verification techniques to generate from these test purposes and the specification, the test cases which will constitute the final test suite.

The combination of these two tools, integrated on top of the efficient *Object*GEODE's simulator, forms the new TESTCOMPOSER tool. It comes as a replacement to the current *Object*GEODE's test generator TTCGEN.

3.2. TESTCOMPOSER principles

TESTCOMPOSER principles are depicted in figure 1. TESTCOMPOSER takes as inputs a SDL/UML specification, a specification of the test environment and a possibly empty set of user defined test purposes. These user defined test purposes can be built interactively using the *Object*GEODE's simulator, or written using MSC observers. The tool can complete the set of test purposes by computation of new test purposes according to structural coverage (arrow 1). Then the generation engine (arrow 2) is fed with each test purpose, and produces test cases which are stored in the test case database. Finally, through the test case database API, the test suite is build and written in a user defined format.

3.2.1. SDL/UML reference specification

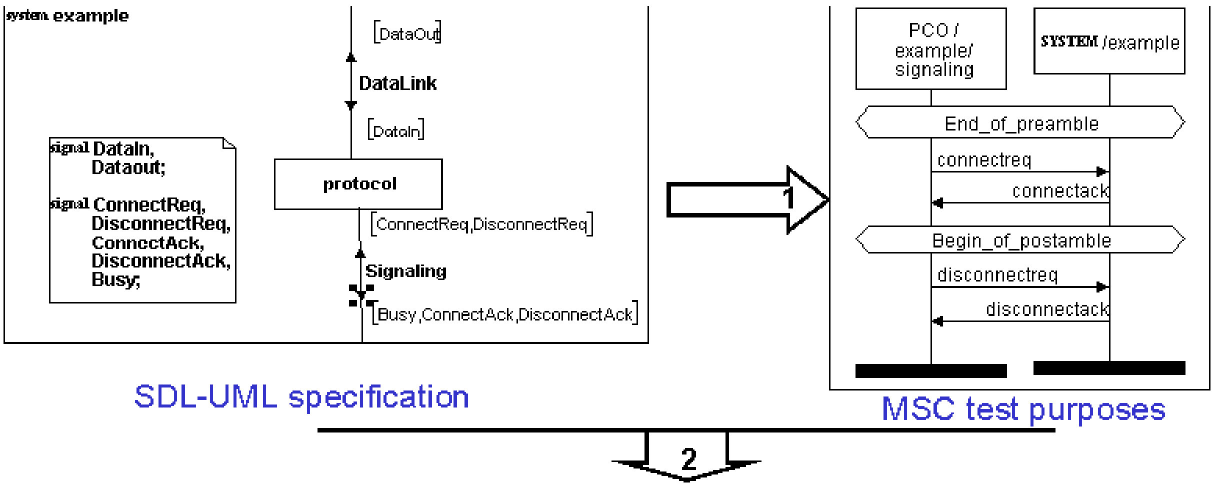
The SDL/UML specification is the central item for efficient test generation. This specification is generally not used for the generation of the code of the Implementation Under Test (IUT), but rather for describing what is the correct behavior of the IUT, from an external view point. In that sense, the specification may not even reflect the internal structure of the IUT.

This specification must be correct with respect to the user requirements, so that it can be used as a reference later during the tests execution. One way to ensure this correctness is of course the use of validation and verification tools.

Test architecture

The specification is also used to describe the test architecture : an IUT is generally not directly reachable, one must interpose communication layers and Service Access Points. Furthermore, more than one tester are sometimes needed for testing distributed systems. In order to model various test architectures, we choose to integrate its description into the SDL-UML specification. A complete specification for test generation thus corresponds to the description of the System Under test : the IUT embedded into its test architecture.

In example 2, the Implementation Under Test *procotol* is described as a SDL `block`. This `block` is connected to the system's environment via two channels, which bear respectively the signals `ConnectReq, DisconnectReq, ConnectAck, DisconnectAck, Busy` and `DataIn and DataOut`. These two channels will correspond to Points of Control and Observation, when we will consider this specifications for test generation purposes.



Test Case Dynamic Behaviour

Test Case Name	:	ex_2
Group	:	
Purpose	:	from state idle of connector(1) receive connectreq send (connectack,) and go to state connected
Configuration	:	
Defaults	:	DEF_0
Comments	:	Generated by test oriented simulation of the test purpose ./ex_tp2.scn

In.	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
0		signalling ! connectreq START TAC			
1		signalling ? connectack CANCEL TAC		(P)	
2		signalling ! disconnectreq START TAC			Start of postamble
3		signalling ? disconnectack CANCEL TAC		P	

TTCN abstract test cases

Figure 1. Test generation process with TestComposer

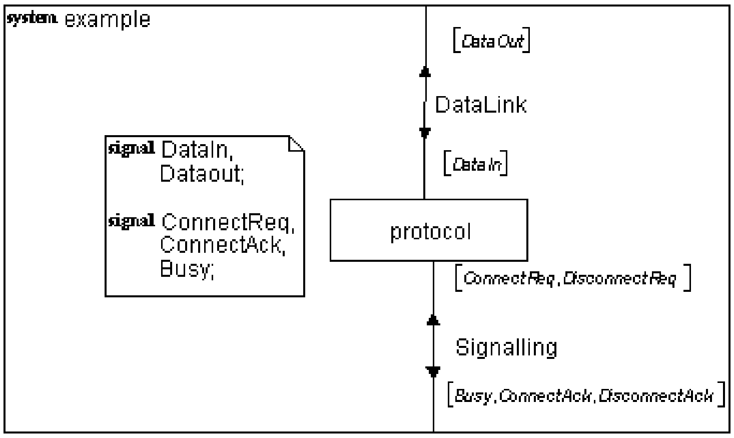


Figure 2. Example of test architecture

Behavior description

At this level, the internal structure of the block *protocol* is hidden. In figure 3, the internal structure, as well as the behavior description of the protocol block is given. This block is composed of two processes : *Connector* which handles the control of the connection, and *Communicator* which handles the data transfer. An internal communication route allows *Connector* to control *Communicator* work.

The behavior of *Connector* is simple : it starts in the `Idle` state; when it receives the `ConnectReq` signal, a check is made to see if resources are available. This check is not formalized, i.e. at this description level, one does not need to know how the check is performed, only that an internal choice is performed. If resources are not available, *Connector* answers `Busy` and returns to the `Idle` state. If resources are available, when *Connector* send the signal `connected` to the *Communicator*, which can begin its work. *Connector* also acknowledges the establishment of the communication by sending `ConnectAck` to its environment.

3.3. Test environment specification

SDL allows to design open systems, i.e. systems whose description is in some sense incomplete, as part of the behavior depends on its interactions with its environment. In the example given in figure 2, the environment and the system interact through the channels SIGNALLING and DATALINK . However, this specification does not say anything more on what the environment does. Will it in fact issue theses messages ? How many times ? In which order ?

To give answers to these questions, TESTCOMPOSER accepts a description of the test environment using the following means :

The `output` command is used to issue one signal just once from the environment to the system.

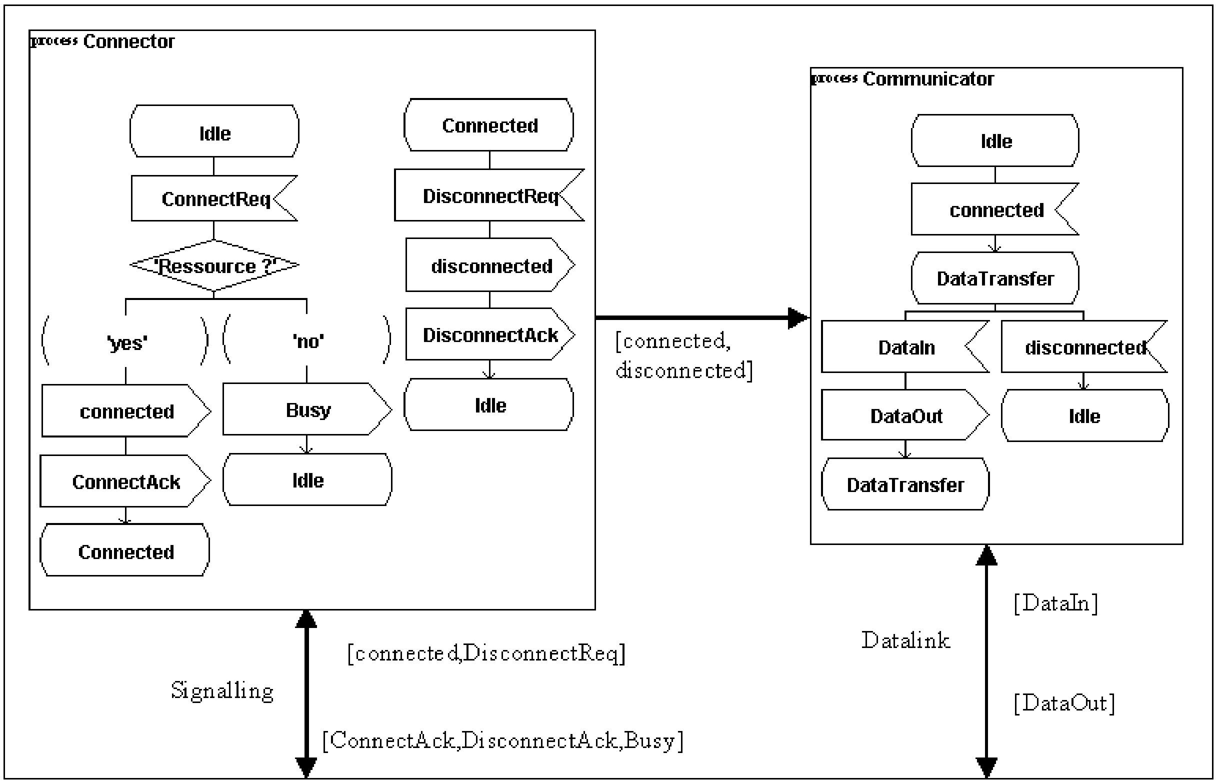


Figure 3. Example of behaviour description

the `feed` command is used to describe what kind of signals the environment is ready to send to the system at any time.

the MSC and GOAL observers come generally in complement to `feeds`, as they are usually used to filter signals so they occur only under specific conditions. For example, a MSC can be used to model the signals interchange of a connection protocol, and then a GOAL [3] observer to model the data transfer. By using appropriate MSC and GOAL observers combination, one can effectively control the simulation.

All of these are parts of the test environment specification. This means in particular that MSC and GOAL observers can be combined with test MSC and GOAL test observers, for a more efficient or more directed test case generation.

For our little example, a possible test environment specification would be :

```
feed Signaling ConnectReq
feed DataLink DataIn
```

which will send these two signals from the environment to the SUT, at any time.

3.4. Test purpose and test case

A test purpose gives a functional view of the test. It is used by TESTCOMPOSER to select in the SDL-UML specification the corresponding behaviors, which will be combined to build the test case. In this respect, the test case can be viewed as the operational view of the test. In short, the test purpose stands for "What should I test ?", where the test case answers to the "How will I test ?" question.

Test purpose description

A test purpose usually is extracted from the system's requirements. It is then formalized, so it can be processed by a tool. A very frequent candidate for this is the Message Sequence Charts (MSC) language [10]: a test purpose MSC contains the main interaction sequence, which correspond to the functionality to test.

An informal test purpose could be "Open the connection, do one data transfer and close the connection". A corresponding abstract, but formal test purpose would be `send ConnectReq, send DataIn, receive DataOut, send DisconnectRequest`, where `ConnectReq, DataIn, ...` are messages send and received by the SUT. A complete test purpose would integrate all the intermediate events needed to fulfill the sequence. Figure 4 shows this kind of test purpose.

TESTCOMPOSER also accepts test purposes described with the GOAL language. GOAL [3] is used to describe automata which will play the role of observer during the system simulation. GOAL observers are used to filter out messages and behaviors, impose a predefined order on messages exchanges, evaluate stop or continue conditions depending on expressions of the system's variables. In addition, GOAL observers have access to all internal elements of the specification, and can also change the values of variables or the contents of queues. So GOAL observers can be used for more elaborate simulations, involving fault injections for example.

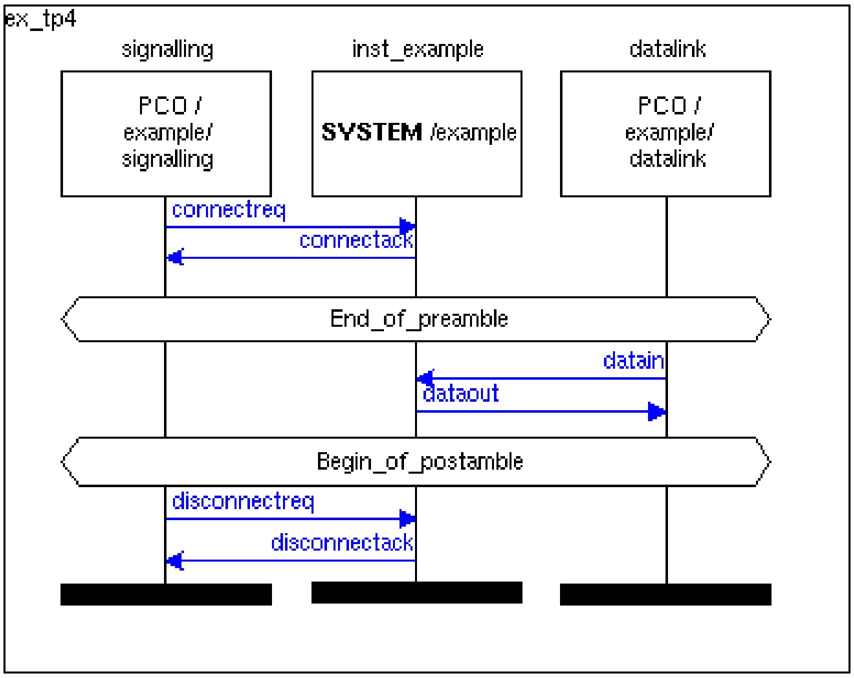


Figure 4. Example of Msc test purpose

"Manual" test purpose description

The test purposes can obviously be written using the Msc and Goal editor which is part of the *Object*Geode toolbox. When an Sdl-Uml specification exists, the simulator allows to produce the Msc test purposes more efficiently, as it will be shown in the next paragraphs. However, protocol norms generally use the Msc language to describe nominal and error handling use cases. These use cases can then be straight used as test purposes.

Interactive test purpose description

When a Sdl specification exists, a more efficient way is to use the *Object*Geode simulator (see section 2). The user can use the simulator to "compute" what is in his sense, an interesting test purpose. Then the simulator can complete this test purpose by computing all the reactions resulting from the last inputs to the system, so that the test purpose ends in a stable testing state. Finally, the simulator saves this test purpose in the test purposes database.

Automatic test purpose generation

In addition to user defined test purposes, TestComposer is able to compute test purposes straight from the system's specification. This feature allows to take the maximum benefits of the specification, especially if its internal architecture maps the IUT architecture, as this is often the case for low level protocols.

The principles of this feature come from a practical viewpoint, which was exemplified by the tool TVEDA : for telecommunication protocols, the SDL specification architecture generally maps to the code architecture. A good test coverage at specification level translates into a almost-as-good coverage at implementation level. So the idea is to generate a set of test purposes such that every transition of the specification is covered by at least one test purpose.

Technically, the generation of test purposes with TESTCOMPOSER is done at a finer level, as the coverage unit chosen is the *basic block*, i.e. roughly the biggest sequence of instructions without branching conditions (no decision, no join, no state change). The computation itself is again based on the *Object*GEODE simulator. We extended its state space exploration techniques so that for each uncovered basic block during exploration, a new test purpose is computed and saved.

In practice, the user first chooses in the specification what are the parts to cover by the test purpose generation, then he gives an upper percentage of coverage to reach (one hundred per cent is usually unreachable for real specifications). Once the computation finishes (by reaching the desired percentage or by resources exhaustion), the user can analyze the coverage results through a graphical browser, which allows to visualize the uncovered parts. Then he can alternatively change some generation parameters, such as the exploration mode, or use his system's knowledge to build interactively a test purpose for the coverage of these parts.

Postamble computation

A test purpose is structured into three parts :

A test body which corresponds to the execution of the events covered by this test purpose,

a preamble which leads the SUT from its starting state to the beginning of the test body,

a postamble which gets the system back in a state where it can execute the next test case. This kind of state can be of course the initial state of the system, but these states usually are no more reachable after the system start. Other states of interest are the *initial stable states*, i.e. the states where the system ends in, when started with no external stimulation.

With TESTCOMPOSER the postamble goal can be user defined in the sense that the user can define the characteristics of the state the postamble should go to. Classical examples can be getting a given process back to its initial state, or forcing the disconnection phase of the protocol. A suitable postamble goal for our example would be the `Idle` state of the *Connector* process. During the test purpose generation process, TESTCOMPOSER automatically computes a path leading to this goal. The postamble of the figure 4 corresponds to a sequence going back to this state.

Test purposes database

The combination of interactive and automatic test purpose generation allows the user to build a test purposes database. This database is saved in session independent files, so the user can possibly go on with the subsequent test generation phases, and come back to add some test purposes in the database.

3.5. Test cases computation

A test purpose describes what is the aim of the test, by giving the sequence of events leading to test success. However, to turn a test purpose into a test case, one must add some more information, especially when dealing with real-time distributed systems. The test case must provide all the information needed by the tester for the test correct execution, but also additional information to provide the user with accurate data in case of test failure. The failure of the test can come from several causes, among them : an implementation fault, a faulty test case, a non deterministic implementation.

Let us take care of the first two cases.

- A implementation fault is what we are looking for. In a real time system, it corresponds essentially to the following situation: a given message was expected during a given interval of time; a wrong message was received, or the right message, but at the wrong time. It is easy to take care of wrong messages, by considering that any unexpected message leads to the test failure. For dealing with out-of-sequence messages, the test case should contain timers to bound the wait for messages.
- We claim that with TestComposer test cases can not be faulty, as long as the Sdl-Uml specification reflects the correct behavior of the specification. This can only be true if this specification has been thoroughly simulated, best verified, so that it can be used as a trustful reference.

The last point is the most tricky one, and in fact it is what makes test generation and writing difficult for distributed systems.

Dealing with non determinism

Distributed and real time systems are generally non deterministic. Moreover, even a deterministic IUT can be viewed as non deterministic when embedded in its test architecture. Many test tools do not allow to handle well non determinism. They usually allow only two test execution results : either the test succeeds, or it fails. However, several executions of the same test can hold both results, when applied on a non deterministic system. This makes the interpretation of test results difficult and more time consuming, as every test failure must first be analyzed to reject failures due to non determinism.

The handling of non determinism in the description of test case thus helps to reduce greatly the test execution costs. To allow this, a test language such as Ttcn, designed for the test of telecommunication protocols, introduces an alternate test verdict, the *Inconclusive verdict*. This verdict allows to distinguish between a test failure due to an internal choice of the SUT, and a real implementation error.

TestComposer is designed to handle correctly non determinism, both during the test case computation and in the test case description. This is explained in the following section.

3.5.1. From a test purpose to a test case

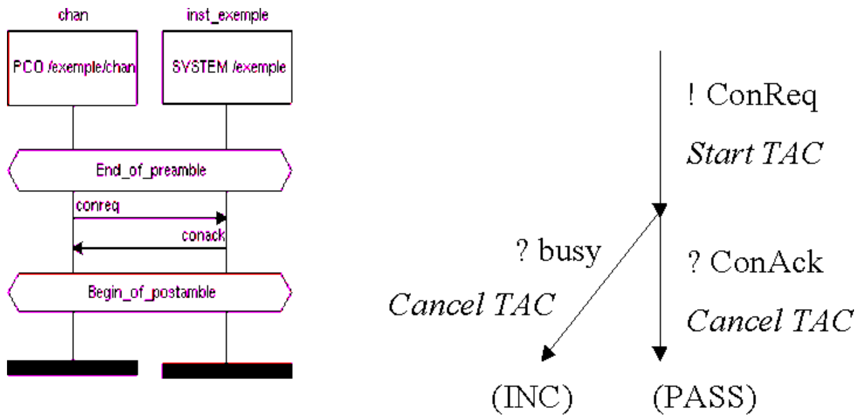


Figure 5. From a test purpose to a test case

The computation of the test case from a test purpose is done in several steps, described in the following paragraphs.

Computation of the main path

The test purpose describes the sequence of events leading to the test success. This sequence can be more or less abstract, i.e. there may be some events missing. In any case, many behaviors described at the specification level can match the test purpose. The first task of the tool is to find such a matching behavior in the specification. This is done by exploration of the state graph which corresponds to all the behaviors described by the specification. The first one found will be the success path of the test case, i.e. the path leading to the Pass verdict. In example 5, the main path is indicated in the test purpose, and transposed as is in the test case, from the tester point of view (i.e. inputs to the system besome outputs from the tester, and vice versa).

Computation of alternatives

Once a success path is found, the tool begins to compute alternatives to this paths. These alternatives are still behaviors described in the specification. They are of two natures :

Pass alternatives When testing distributed systems, one often has to wait for two or more messages, without any idea of the outgoing order of these messages. Even if these messages are issued in a given order by the IUT, the use of several PCOs and communication delays between the IUT and the tester can change the order of messages on reception. So even if the test success path specifies that message `a` should be followed by message `b`, the other order can lead to the same test success.

In order to detect these situations, TestComposer explores alternatives paths from the success path, to see if they also lead to success. These paths are explored as long as they consist only of SUT's outputs. Loops are also computed, i.e. paths going back to some previous step of the main path.

Inconclusive alternatives If an alternative to the main path does not lead to the success, it still remains a valid sequence, as it is described in the specification. So TestComposer keeps them in the computed test case, truncated to their first event, and annotated with the *Inconclusive* verdict.

In example 5, an *Inconclusive* alternative is computed, for the reception of the valid, but not desired message `Busy`.

Timers setting

Several timers are introduced to handle the different situations the tester must face :

"Wait for answer" timer TAC This timer is the most classical one. The tester sends a message, then waits a bounded time for reactions. Its timeout leads to the test failure.

"Wait for user input" timer TWAIT Protocol testing sometimes requires user interaction, even in the course of an automatic test execution, as The tester does not always have access to all the interfaces of the System Under Test. In that case, a long range special timer is set, leaving enough time for a user to provide the necessary inputs. Its timeout leads to the test failure.

"Wait for no answer" timer TNOAC A test sequence can lead the system in a state where it just waits for another input, without emitting any message to the tester. In order to detect this situation, the tester sets a specific timer, which timeout allows to continue the test.

"Wait for timer expiration" timer TEMPTY The System Under Test can have its own timers running. The timer TEMPTY is meant for the detection of the expiration of these timers.

3.5.2. On the fly generation

The generation engine is base on state space exploration techniques. Many test generation tools need to build the complete state space before being able to compute a test case. The ability to work "on the fly", i.e. generation of the test case during the state space exploration is a key to efficiency, as it allows to stop the exploration as soon as the test case is complete

3.6. From test cases to the test suite

The test cases produced are stored into a database. This database is accessible through an Application Programming Interface which provides access to data types and signals definition, messages values, test architecture, test cases and test suite definitions. Then a test suite production module can be interfaced to this API, to produce the user desired format for its test suite.

The first test suite production module we implemented with this API generates a Ttcn-mp file. Ttcn is an international standard for the description of protocols tests, defined in [1]. Existing Ttcn environments allow the definition and edition of test suites, as well as the compilation and execution of the tests on real targets.

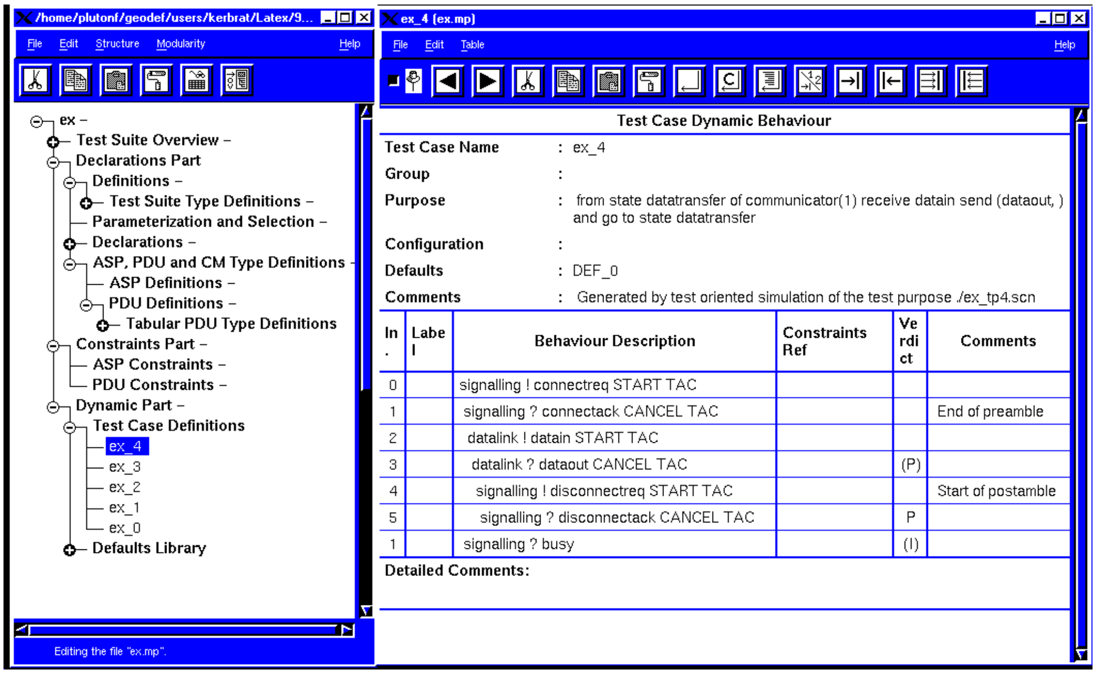


Figure 6. Excerpt of the TTCN suite, with Sema Group TED editor

On the TTCN example of figure 6, we show the final test case, generated from the figure 4. TTCN is a tabular notation, the indentation (denoted in the first column) indicates the sequence number of the corresponding event. Each event is either a sending (!) or reception (?) of a message, on a given PCO. Timer commands (`START TAC`, `CANCEL TAC`) are associated to each event.

The main sequence is annotated with the temporary and final Pass verdicts, and information about its structure. The `busy` reception is indicated as an alternative to the `ConnectAck` reception, and leads to the temporary Inconclusive verdict.

Finally, an informal test purpose is synthesized from the test body, allowing a human operator to manually perform the test if necessary.

3.7. TestComposer added-value

The added value of TestComposer depends first on the availability of a formal specification of the SUT. Developing a SDL-UML specification is a fruitful task, as it allows to track many analysis and design errors at a early stage, just by having to formalize the requirements, and later by using a tool such as the simulator to simulate and validate the specification behaviors. Writing and maintaining such a specification is very cost effective, especially when dealing with new systems. In this context, test generation as well as code generation brings its best value.

For existing systems, where no formal specification exists, the cost of setting up automated test generation must be compared to the cost of writing a test suite from scratch.

The specification must be written, validated, the test environment must be described and all generation parameters must be set and tuned. The complete task can take up to twice the time needed to write the test suite, especially if we consider someone expert in test languages and novice in specification languages, as shown by the ongoing study [8]. However, this overhead is paid for only initially,furthermore, one must consider the other benefits of test generation :

Link between requirements and tests The relation between tests and high-level requirements is better maintained, as the SDL-UML specification comes straight from the high-level requirements. Then TESTCOMPOSER test generation process ensures the correctness of the test produced, and allows to trace each test back to the specification.

Easy test suites maintenance Most requirement or design changes usually imply major rewriting of the test suites. When using automated test generation, any requirement or design change can be reflected directly in the SDL-UML specification, and the new test suites are then re-generated at no cost.

Faster implementation debugging the SDL-UML specification, once validated, can be considered as a reference. Then TESTCOMPOSER ensures that the tests generated are correct, as long as the specification is correct. It is a major improvement on current test practices; when a test fails, the usual question is "What is wrong, the implementation, or the test ?" As up to 15 percent of hand written test cases can be false, this is a valid, very time consuming question. With TESTCOMPOSER, the specification is the reference, and the test failure is always attributed to the implementation.

Identifying `Inconclusive` verdicts from `Fail` verdicts is also a gain when debugging, as an `Inconclusive` does not require an investigation.

4. Executing tests with TESTPLAYER

TESTCOMPOSER requires a full SDL- UMLspecification in order to generate tests. However, it is often convenient to use directly the MSC language to describe and execute tests, without having to bother with the full system's description.

This is where a tool such as TESTPLAYER brings its value. TESTPLAYER defines an open architecture for executing conformance tests, and consists of a set of tools for interpreting test MSCs, translating MSCs into some internal test scripting language, executing test scripts and examining test results. TESTPLAYER can take as inputs the test MSCs generated by TESTCOMPOSER, or independently, test MSCs provided by some other means.

The architecture of TESTPLAYER described below, relies on two fundamentals: the MSC language and an internal scripting language called TDL (Test Description Language). The TDL language exists in order to provide the parts that are missing in MSC but are still necessary to express certain test cases: variables, control structures, explicit verdicts and an imperative language. The TDL language is based on a set of test primitives (see [20] for a complete description):

Output - sends a message to the system

Expect - generically specifies a signal that is expected to come from the system

TimerSet, TimerReset, TimerWait - for manipulating timers

Alternative - expects one in a set of possible (generically specified) responses from the system

Verdict - explicitly gives the verdict of a test script

These primitives can be implemented in almost any language (C/C++, TCL, Java) at little expense, and the host language brings in variables and control structures. Within our architecture the main host language is TCL [13] and TDL denotes the extended TCL.

TESTPLAYER provides an open architecture for testing, based on the lightweight formalisms of tMSC and TDL. It offers customization and power of expression sufficient to be applicable to a large number of real systems and test cases.

Figure 7 depicts the overall architecture of TESTPLAYER. We distinguish between three phases: the test preparation phase, the test execution phase and results use phase.

Test campaign preparation

When preparing a test campaign, the user must provide the test cases. A test case may either be expressed by an MSC hand-written or generated by TESTCOMPOSER or it may be expressed directly in a TDL script. If the former is the case, then the MSC is translated automatically in a TDL script, which can be modified further. The TDL Gen tool that does this translation is provided as a part of TESTPLAYER. The rules for translating an MSC into a TDL script are quite straightforward, as MSC constructs map almost one-to-one to TDL primitives.

The engine that executes the TDL scripts needs to know the data types and the signals that will be exchanged between the tester and the system under test (SUT), in order to be able to encode/decode and send them out to the system. Consequently, a part of the preparation phase is dedicated to the description of the data and signal types of the system, and to the generation of the encoding/decoding component (1) of the TDL execution engine. The data and signal types may be described in SDL [16] and ASN.1 /citeZ.105, and then the encoding/decoding component (1) is generated automatically using StubGen, a tool provided with TESTPLAYER.

Test campaign execution

At execution time, the core of TESTPLAYER is the TDL execution engine. The library containing the implementation of the test primitives is the fixed part of execution engine, i.e. it cannot be modified or replaced. The entire conception and architecture of TESTPLAYER relies on it.

The test primitives library depends on a set of customizable components. First there is the TCL wrapper that makes the test primitives (Output, Expect, Verdict, etc.) available as TCL commands in an extended TCL interpreter. This is essential, since the TDL scripts generated from MSCs are TCL programs (with calls to our primitives). As mentioned above, the implementation of the test primitives is not depending on any host language,

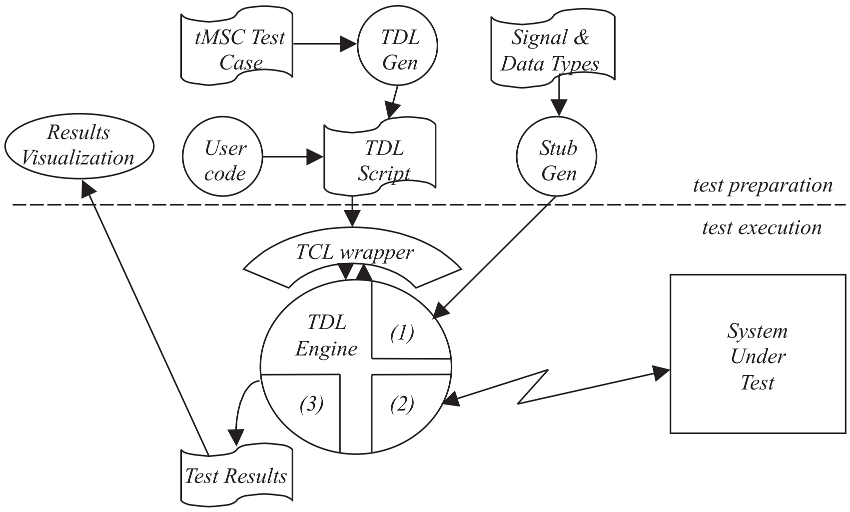


Figure 7. TestPlayer Open Architecture

so they may be used as well from C. However, by doing so, the user misses the advantage of having an automated translator that converts test Mscs into Tdl scripts.

Another component used by the test primitives implementation, and part of the Tdl engine, is the Signal Encoding/Decoding component denoted by (1) on Figure 7. This component contains call back functions for signal and data encoding/decoding/matching. The tester communicates with the SUT using a certain protocol, 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 or Alternative command. As mentioned above, the Signal Encoding/Decoding component 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 7 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 means. The Protocol component (2) must provide a simple interface containing functions for opening and closing connections, sending binary data to a connection and doing 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 the 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.

Using test results

After executing a test campaign, the results of the test cases and the events that occurred all along the campaign can be visualized. Using a specialized tool, the Results Visualization tool, one may navigate from any event occurrence to the TDL source line or to the MSC construct that corresponds to it. This tool accesses the information contained by the test log, using a replaceable component for reading the log, similar to (3).

Thus our test execution architecture is based on the interpretation of test MSCs and on the Test Description Language, and 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. We instantiated the open components to obtain a concrete test execution framework for testing *Object*GEODE systems.

4.1. TESTPLAYER for *Object*GEODE

TESTPLAYER for *Object*GEODE is the complete TESTPLAYER framework for systems modeled in SDL [16] with *Object*GEODE [19] and for which code is automatically generated with the *Object*GEODE SDL-C Code Generator. It is an instantiation of all the generic components described in the previous section. The Signals Encoding/Decoding component (1) is generated by the StubGen tool, which is able to take in input the SDL description of the system and extract all the necessary information. The Protocol component (2) uses TCP/IP to communicate with the system side. The tested system being generated automatically using the *Object*GEODE SDLC Code Generator, a proxy task is automatically added to the system. This task ensures the communication with the tester using the same protocol as the TDL engine component (2), and has a minimal interference with the implementation under test. Finally, the Test Reports component (3) was implemented so that it keeps the test results in an ASCII format test log. These results can be visualized and used with the Results Visualization tool.

5. Conclusion

We have presented TESTCOMPOSER, an automated test generation tool for conformance testing. It is completed by TESTPLAYER, an open and flexible environment for test description and execution using MSCs. Both modules will be part of the *Object*GEODE toolset during the coming year.

TESTCOMPOSER allows the test designer to work on a high level SDL/UML specification, rather than working at a low level test language. This specification can be validated, using a tool such as *Object*GEODE's simulator. This allows to consider this specification as a reference. TESTCOMPOSER then provides for assisted or automated test generation, dealing completely and correctly with the non deterministic nature of distributed real-time applications.

The ability to generate test purposes based on the structural coverage of the specification makes of TESTCOMPOSER an important tool for certification purposes. If we consider a critical application, which code is generated from an SDL-UML specification, then its certification requires either the certification of the code generator, or a complete, one hundred per cent functional and structural coverage of the code by the tests. As the code generated from SDL is structurally very close to the specification itself, test purposes generated by TESTCOMPOSER are very useful to ensure coverage of the code. Furthermore, abstract test purposes can be defined by the user, to provide for functional coverage.

Last but not least, the performances of the test case generation engine allow the computation of long, tricky test cases which would be hard to design by hand. Failures of critical systems usually don't come from single errors, but mostly from the conjunction of different errors. Writing tests for this kind of "errors interaction" is hard and very time consuming; the ability to work on-the-fly brings here the possibility to explore very long behaviours, and to generate test cases corresponding to these behaviours.

TESTCOMPOSER is actually tuned for working in a TTCN environment, together with the TED editor of the UK-French company SEMA GROUP, and the TTCN compilation execution toolbox from the German company DANET GMBH. However, its reliance on standards for test purposes description, and its API to the test suite make it open to user specific languages and testbeds.

The main bottleneck for the use of TESTCOMPOSER is writing and validating the SDL-UML specification. Development processes which do not already integrate such practices can find this is too heavy an investment to make in the short term. Using a light weight approach such as TESTPLAYER, which in its most generic form does not require a full SDL-UML specification, can fit these development processes.

REFERENCES

1. ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - Part 2 : Abstract Test Suite Specification - Part 3 : The Tree and Tabular Combined Notation (TTCN), 1992.
2. ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework, 1992.
3. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *SDL forum'95*. Elsevier Science (North Holland), 1995.
4. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, Special Issue on The Application of Formal Methods in Industry Critical Systems*, To appear, 1999.
5. E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.
6. M. Clatin, R. Groz, M. Phalippou, and Richard Thummel. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, *Proceedings of IWPTS'95 (8th Int. Workshop on Protocol Test Systems, Evry, France)*. INT, sep 1995.
7. E. Cohen. UML, SDL, and MSC: Object-oriented and real-time combined. *Distributed Computing*, 1(2):146–160, June 1998.

8. CS-Telecom. Experimentation of an automated test generation prototype. Technical report, CS-Telecom, 1999.
9. L. Doldi, V. Encontre, J.-C. Fernandez, S. Jéron, T. and¡ Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9^{th} International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1996.
10. Jens Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts, Ph.D. Thesis*. Universitat Bern, 1994.
11. A. Kerbrat, T.Jéron, and R. Groz. Automated test generation from SDL specifications. In *Proceedings of SDL forum'99*. Elsevier Science (North Holland), 1999.
12. I. Ober and A. Kerbrat. Test specification and execution using tmscs. In *Submitted to Forte/PSTV'99*, 1999.
13. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Pub Co, 1994.
14. M. Phalippou and R. Groz. Evaluation of an empirical approach for computer-aided test cases generation. In I. Davidson, editor, *Proceedings of IWPTS'90 (3rd Int. Workshop on Protocol Test Systems, McLean VA, USA)*. Corporation for Open Systems, oct 1990.
15. Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.
16. ITU-T Recommendation Z-100. Specification and Description Language, 1996.
17. ITU-T Recommendation Z-120. Message Sequence Charts, 1996.
18. ITU-T Recommendation Z.500. Formal methods for conformance testing, 1997.
19. VERILOG. ObjectGeode reference manuals. Technical report, VERILOG, 1996.
20. VERILOG. TestPlayer Reference Manual. Technical report, VERILOG, 1999.