A pragmatic approach to test generation

E. Pérez, E. Algaba, M. Monedero
Telefónica I+D
Emilio Vargas, 6
28043 - Madrid (Spain)
emilla@tid.es, algaba@tid.es, mmr@tid.es

Abstract
This paper presents a pragmatic approach to the problem of the automatic generation of a test suite for a given system. It introduces the GAP tool, embedded in the HARPO toolkit, which is capable of generating TTCN test suites starting from a SDL specification of the system and test purposes written in MSC notation. In addition to this, GAP computes coverage measures for these tests, which represent an evaluation of their quality.

Keywords
Test generation, SDL, MSC, ISO 9646, TTCN, coverage

1 INTRODUCTION

The ISO conformance testing methodology, ISO 9646 [7], is widely accepted as the main framework in telecommunication systems testing. This methodology includes general concepts on conformance testing and test methods, the test specification language TTCN [8], and the process of specifying, implementing and executing a test campaign. The major point in a test process lies on the availability of a test suite, which must be closely related to the system specification. Unfortunately, the manual design of a test suite is an error prone, time and resource consuming task.

The use of FDTs, especially SDL [1], in system and protocol specifications establishes a suitable environment for the development of automatic test generation tools. These tools, whose main input is the formal specification of the system, help solving the problem of a manual test suite production. Furthermore, the automatic nature of the process ensures the correctness of the generated tests and eases the computation of test quality measures, namely coverage.

GAP, embedded in the HARPO toolkit [6] for the development of testing tools, represents a practical approach to these automatic test generation tools. It focuses on SDL system specifications, test purposes described using MSC[3][4] notation and test cases written in TTCN. In order to generate the test cases, the GAP tool simulates the behaviour of the system under test. The simulation is guided by the MSC test purpose throughout the entire generation process. This approach takes advantage of the increasing number of SDL formal system specifications available to derive test cases without further interpretations of the standards. Moreover, an executable test suite can be implemented with these test cases in an automatic way using the remaining tools within the HARPO toolkit. The whole process of developing a testing tool is thus greatly automatized, being feasible to obtain an executable test suite starting from the formal specification of the system in just one process.

The purpose of this paper is to describe the GAP tool (described in section 3) in its environment. Section 2 describes the methodology chosen in GAP and the tool architecture.

Even though references are made to ISO 9646 methodology in this paper, the GAP automatic test generation tool does not restrict its output to conformance tests.

2 GAP METHODOLOGY

Conventional test generation methodologies can be classified into two main categories:

- Computer Aided Test Generation techniques (CATG): the tests are obtained in a semiautomatic manner, via an interaction of the user with the formal specification of the system. They are usually based on simulation techniques.
- Automatic Test Generation techniques (ATG): the tests are automatically generated by a program that exhaustively explores the behaviour of the system under test, represented by a FSM derived from its formal specification.

Both approaches have their pros and cons.

The test generation methodology chosen in the GAP tool combines both techniques in order to take full advantage of their positive features while minimizing their respective problems. GAP makes use of ATG techniques in the sense of exploring the behaviour of the system so as to generate tests. The difference is that the behaviour of the system is not exhaustively explored, but guided by a test purpose specified by the user. Thus, the ATG state explosion problem is avoided, because the explored behaviour is only the subset of the behaviour of the system that verifies the test purpose.

As we just said, ATG techniques are usually based on a FSM derived from the specification of the system, producing a test suite skeleton (behaviour) which has to be manually completed. GAP uses CATG simulation techniques to explore the behaviour of the system, thus producing complete test suite specifications (behaviour and data). Moreover, coverage measures can be computed over the original system specification, avoiding the need to keep links between the FSM and the original specification. Another advantage of GAP methodology is that the generated test suite is structured in the way it is expected to be, due to the use of user specified test purposes, and the test suite is kept down to a manageable size, unlike the one generated with ATG techniques.

To sum up, the inputs to GAP methodology are the formal specification of the system and a set of test purposes, which enables it to produce a complete test suite specification and coverage measures related to the original system specification.

HARPO is a test tool development environment including a TTCN to C compiler (behaviour and data), PICS and PIXIT proforma editors and a testing tool operation environment. Integrating the GAP tool in the HARPO toolkit provides a high degree of automatization in the test tool development and operation process. Such an environment minimizes the problems of manual test suite production, but its complexity does not completely disappear. It is shifted to the specification of the system and the selection and specification of the test purposes. Nevertheless, this methodology provides great advantages compared to the manual specification of tests: reduced costs and increased quantity and quality of generated tests.

2.1 Methodology elements

The elements of the GAP methodology and the languages used for their specification are presented below.

System under test

The system under test is specified using SDL-92. Nowadays SDL is the most accepted FDT in telecommunication industry, has the largest commercial tool support and is also being used by standardization organizations (ITU, ISO, ETSI).
GAP can process either ACT ONE or ASN.1[5] (following Z.105[2] recommendation) data type specifications.

Test purposes

Test purposes in GAP are used to specify behaviour skeletons that drive the test generation, avoiding the simulation of correct test behaviours that are not useful for a given purpose. MSC notation has been extended with annotations to ease the specification of test purposes. A MSC is no longer interpreted as a complete trace of an execution of the system, but as an open pattern that lets the tool derive a subset of the system behaviour that fits it. Thus, a test purpose in GAP is used to generate several test cases.

Test suite specification

The generated test suite, ATS (Abstract Test Suite) according to ISO 9646 terminology, is written in TTCN (MP format). This is a complete test suite including overview, data types, constraints (data values) and behaviour. It can be processed by the TTCN compiler included in HARPO, thus producing an executable version of the tests, ETS (Executable Test Suite) according to ISO 9646 terminology.

Coverage

An important quality measure of a test suite is the coverage degree of the system it provides. The goal is to select the appropriate measures from a practical point of view. GAP computes state, signal and transition coverage measures. Another interesting concept is incremental coverage. The quality of two test suites can be compared with this measurement.

2.2 Work methodology with GAP (HARPO)

Figure 1 depicts the tool development and operation process using GAP within the HARPO toolkit.

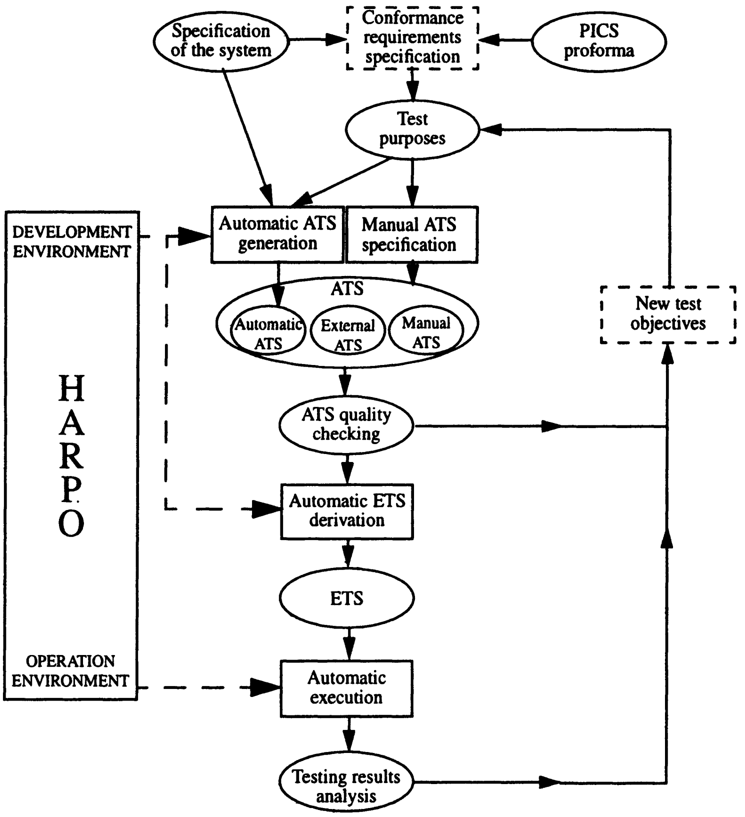

Figure 1 Work methodology with GAP (HARPO).

The process comprises several steps:

1. Identifying conformance requirements. This task requires a previous knowledge of the system under test.

2. Obtaining a SDL formal system specification. It may be obtained from external sources such as ITU, ISO, ETSI, etc. Otherwise, this specification will have to be produced manually.
3. Development of test purposes for the conformance requirements.
4. Execution of the GAP tool taking as inputs the specification of the system and test purposes, to produce a TTCN test suite (ATS).
5. Analysis of the generated test suite to check that it meets the expectations. In case of failure, new test purposes may be defined covering requirements that were not tested. It is possible to manually modify the ATS.
6. Compilation of the test suite with the TTCN compiler included in HARPO, to produce an executable version of the test suite.
7. Execution of the tests against the system and analysis of the results. This step may drive the user to the definition of new testing purposes.

The methodology defined above is in accordance with that defined by the ISO 9646 standard. The main advantage achieved using GAP tool within HARPO is the high degree of automatization in the process of developing a testing tool, starting from a formal description of a system and obtaining an executable test tool.

3 GAP TOOL ARCHITECTURE

Figure 2 depicts the architecture of the GAP tool. There are three main blocks: syntactical front-end (block 1), test generator (block 2) and data type translator and data values validator (block 3).

3.1 Syntactical front-end

This module, block 1 in figure 2, reads input data and stores it in memory in a suitable format to ease navigation for simulation and data type translation purposes. The specification of the system is written in SDL/PR language while test purposes are specified in MSC/PR notation. There are several commercial SDL and MSC graphical editors (SDT, GEODE, etc.) that can be used to dump these specifications from graphical to textual format.

The goal of the module is to store in memory a representation of the specification of the system and the test purposes, which is a well known problem amongst compiler developers. The final result is a memory structure known as abstract syntax tree (AST). Each node of this tree stores a SDL construction and its associated information. An application programming interface (API) is supplied to ease the accesses performed by the other modules. This API provides:

- Easy navigation throughout (the behaviour of) the system to derive test cases, guided by the test purposes.
- Capability to use the AST to store additional information produced while generating tests.

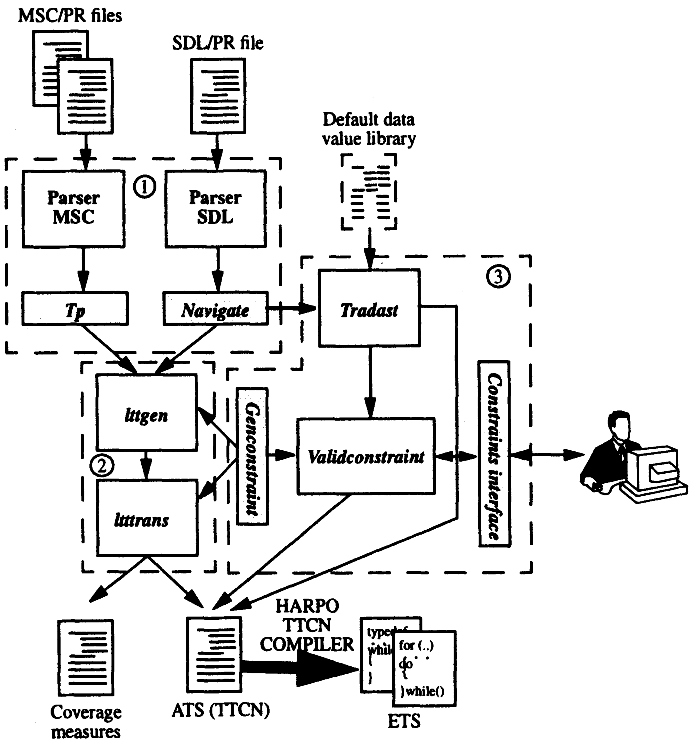

Figure 2 GAP tool architecture.

3.2 Test generator

The test generator, block 2 in figure 2, comprises two modules: lttgen and ltttrans.

The lttgen module simulates the specification of the system as stated in the test purpose and generates an intermediate structure called labelled transition tree (LTT).

The LTTs, the algorithm used by the generator and LTT translator and coverage measures computing module are described below.

Labelled transition trees

A LTT is a data structure that symbolically represents the evolution of a system. The LTT is a behaviour tree composed of nodes and arrows. Starting from a root node, it does not contain backward links (no cycles) nor subtree sharing (it is not a graph). Each node represents a state of the system and each arrow a transition. One LTT is generated for each test purpose, and it comprises the activity of the SDL objects: processes, channels and queues.

The LTT is dynamically built during the simulation, then transformed in its mirror image to reflect the point of view of the tester (see figure 3) and finally dumped in behaviour and data predicates.

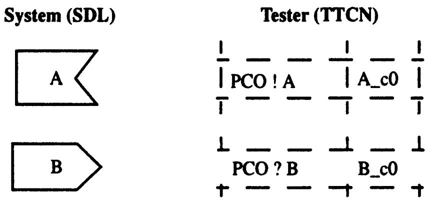

Figure 3 System-tester duality.

Test purposes

The MSC notation is a trace language that models the observational behaviour of a system. Its design shares a common philosophy with SDL.

GAP uses MSCs to describe test purposes. Extensions have been added to the MSC notation to allow for a better description of test purposes. This extensions are called annotations and are inserted in the comments part of MSCs. At a graphical level they only represent part of the documentation of the generated tests. For processing purposes, they are used to limit the exploration of possible behaviours while simulating the system. The goal of annotations is to keep the generated LTT down to a manageable size. Some of the most characteristic annotations are: maximum number of signals in channels, maximum number of process instantiations, maximum depth between two signals, maximum number of occurrences of a signal, signal disallowing, preamble and postamble annotations, etc. Useless loops are avoided via annotations. The user decides which loops are useless, and annotates the test purpose accordingly.

An example of test purpose is depicted in figure 4. It was used to test the call diversion service of the telephone network. This service allows the user to redirect incoming phone calls to another phone number. On the left side of the figure, an example of test purpose for this service is shown. The test purpose is not a complete trace of an execution of the system. Thus the MSC is interpreted as an open behaviour pattern, and several test cases will be generated for it. In fact, every possible test behaviour that fits in the test purpose will be generated. Two of the generated test cases are illustrated on the right side of the figure (MSC notation is used for test cases instead of TTCN in order to simplify it).

Test case 2 contains a loop (Pick, ..., Hang, Pick, ...). The user is responsible of determining if the loop in Test case 2 is useless or not. Such loops may be useful in some situations, i.e. while testing the data transfer phase of a protocol. The generation of these loops is controlled via annotations: signal disallowing, maximum number of occurrences of a signal, etc. For example if the user decides not to generate test cases with Pick loops, the test purpose can be described as in figure 5, where a disallowing signal annotation for Pick has been added. This test purpose will generate Test case 1 but not number 2.

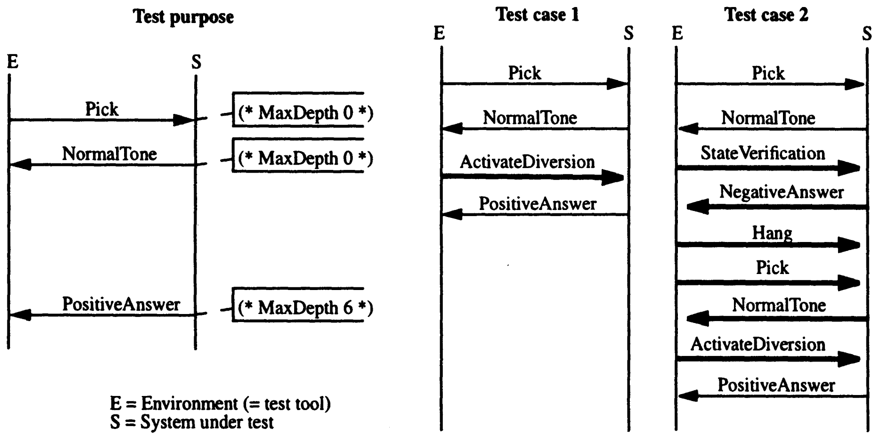

Figure 4 Test purpose and two generated tests for it.

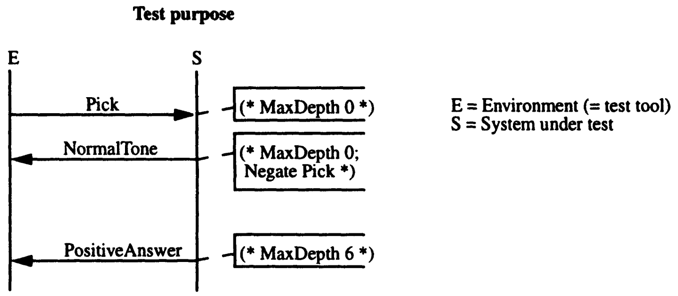

Figure 5 Modified test purpose.

As stated before, the test purpose constitutes a mere behaviour pattern, and not a complete trace of the behaviour of the system.

LTT generation

A symbolic simulator or SDL machine is used to generate the LTTs. Starting from the initial state where processes are at their start point and queues and channels are empty, the SDL machine executes the specification generating the allowed transitions. The state of the system (variable values, processes state, queues and channels state and active timers) is stored in the LTT nodes, whilst those events that produce a state change are stored as transitions. Some of these changes are determined by the appearance of a signal in an input channel from the environment, that is to say, an input transition. On the other hand, the disappearance of a signal in a channel pointing to the environment constitutes an output transition. The remaining transitions are merely internal. The LTT is generated following the rules stated in the test purpose (one LTT is generated for each test purpose). Starting from the initial state, all the possible evolutions of the system are computed and carefully stored in the LTT. Those branches that do not verify the test purpose are pruned during the generation. Thus, the generated LTT represents the behaviour subset of the system that is significant for the test purpose.

Predicates on data are gathered during the generation by means of decision clauses in SDL. In the example in figure 6, if the generation evolves through the TRUE branch, an associated predicate a>=5 will be stored. All these predicates are dumped out to the data validator module, whose purpose is to find the appropriate data values that satisfy them. Subsection 3.3 describes this module.

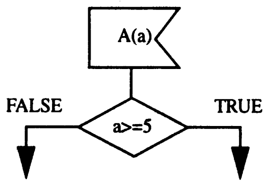

Figure 6 Decision clause.

It is important to state that after the LTT has been generated, only those events that can be observed from the environment are taken into account, namely external signals and timers. Those transitions with no reflection in the environment are pruned, thus simplifying the LTT.

LTT transformation

The LTT reflects a subset of the possible behaviour of the system, but it vaguely resembles a list of TTCN test cases. The final goal is to generate TTCN so several transformations must be applied.

First of all, the direction of send and receive events must be inverted to reflect the point of view of the tester (see figure 3).

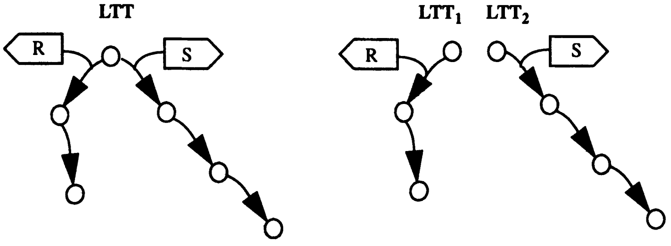

Figure 7 LTT splitting.

Secondly, the LTT must be split into a list of test cases. In the example in figure 7, there are two send events at the same level (from the point of view of the tester). The tester can send either one or the other but never both. The LTT is therefore split in two, LTT_1 and LTT_2, from which two test cases will be derived. Finally, once the initial LTT has been inverted and split into smaller LTTs (test cases), these are dumped in TTCN.

Coverage

The ltttrans module is also in charge of computing coverage measures. It counts the number of states, signals, timers and transitions the LTT covers. A distinction is made between observable signals (black box coverage) and internal ones (white box coverage). The GAP tool calculates state, signal and transition coverage measures.

State coverage is a white box measure. It is a weak measure, because there are usually many different paths to go from one state to another. Therefore the generated tests can cover all the states without completely exercising the system. Thus, the usually accepted value for this measure is 100% of states covered.

Two signal coverage measures are computed: black box and white box. These measures are more demanding than state coverage measures. Nevertheless, 100% coverage is required for them as well. The reason is that they do not take into account the dynamics of the system (the sequence in which the signals are ordered).

Transition coverage, a variant of branch coverage, is a white box measure. It is the stronger measure computed by GAP. The accomplishment of 100% depends on the absence of dead (not reachable) text in the specification and the feasibility of finding values to satisfy all the predicates. Working values near 100% are usually accepted.

Apart from these absolute measures, GAP can also compute an incremental measure of the coverage achieved by the tests generated for one test purpose with respect to those generated for another one. This incremental coverage gives an estimation about what is being tested with a set of tests that has not been tested with another one. Suppose there are two sets of tests for the same system, A and B. If the incremental coverage of B with respect to A is zero, this implies that B is not checking anything that has not already been tested by A. In such a situation, the tests included in B can be discarded. GAP computes state, signal and transition incremental coverage.

3.3 Data type translator and value validator

It corresponds to block 3 on figure 2, and comprises two modules: Tradast and Validconstraint.

The Tradast module translates signals and data types appearing in the specification of the system to a valid notation in TTCN, either ASN.1 or tabular data types.

The Validconstraint module displays the constraints values needed to complete the tests and their associated predicates, if any. An external data value library supplied by the user can be read by the tool in order to ease the process of filling in the constraint values. This library is always dependant on the system under test. Moreover, the values introduced by the user are syntactically and semantically checked, and the corresponding predicates are validated.

Data type translator

This module is responsible for the translation of the signals in the SDL system. It also translates ACT ONE data types to ASN.1 or tabular types, and directly dumps those types already written in ASN.1 (Rec. Z.105). Signals are translated into ASPs and

PDUs, and data types into TTCN types, either in ASN.1 or tabular declarations.

SDL uses ACT ONE notation to define abstract data types. An ACT ONE data type is determined by:

- Literal values
- Operators for data types, defined by their signature
- Axioms defining the semantic of the operators

A data type containing these elements is called a partial type description in SDL.

ACT ONE operators are used as patterns to translate data types. The translation is carried out by applying several heuristics that establish parallelisms between the ACT ONE data type definition and its TTCN type translation. The tool tries to identify these parallelisms and, if it succeeds, executes the translation.

Some predefined data types in SDL can be directly translated, due to their semantic equivalence. GAP needs heuristics to translate the remaining SDL predefined data types.

Data value validator

The function of this module is to supply correct data values (constraints in TTCN terminology) that verify the predicates collected during the simulation, and to dump them into TTCN constraints (ASN.1 or tabular).

Send and receive constraints are computed in a different manner.

A reception constraint in the tester is derived from a send event from the system to the environment (see figure 8). A send event in the system must be completely specified based on constants and variables of the system. Therefore, the GAP tool knows the exact value of the constraint that must be generated. If this reception constraint depends on the value of a variable which cannot be solved, the evolution of the variable is dumped within the TTCN behaviour and passed as a parameter to the constraint (see figure 8).

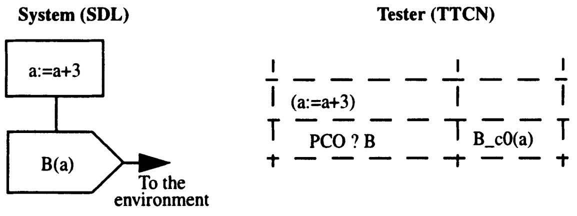

Figure 8 Signal to reception constraint translation.

Send constraints in the tester are derived from reception events in the system (see figure 9). These values are received from the environment, so they are not known at generation time. Therefore, these values must be filled in during the final test value validation phase. Values may have associated predicates: for instance, in figure 9, if the test case evolved through the TRUE branch in the decision clause, the tool would need a constraint A_c0, whose first field (because a is the first parameter of signal A) should be equal or greater than 5.

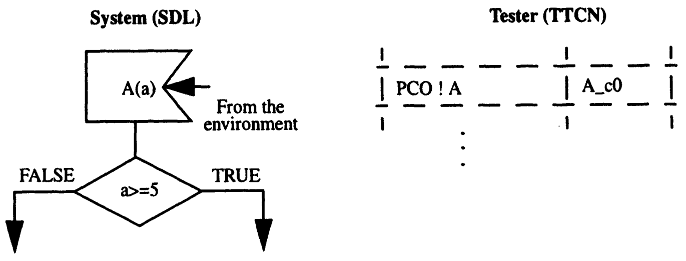

Figure 9 Signal to send constraint translation.

There are two data value generation modes in GAP: semiautomatic and automatic. In the semiautomatic mode, the tool automatically generates all the constraints with no associated predicates, leaving to the user the task of filling in those constraints with associated predicates. The generated values either come from an external default data values library or have been automatically generated choosing any valid data value for each constraint field according to its type definition. The tool also helps the user to fill in the constraints with associated predicates by means of suggesting default values from the external library. Once the constraints have been filled in, the tool checks that the supplied data verify both their type definitions and their associated predicates.

When working in automatic mode, the tool fills in all the needed constraints, either with default values from the library or with generated values. No value checking against associated predicates is performed in this mode. This mode is useful in the first development phases of the test specification, when the user is focused on checking if the dynamic behaviour of the generated tests fits the initial testing goals for the system.

4 CASE STUDY FOR THE TPO PROTOCOL

In this section a brief example of the generated TTCN, for OSI Transport Protocol class 0 (TP0), is introduced. The test purpose is specified in figure 10. One of the generated test cases and its default are depicted in figure 11: *nconreq*, *nconcnf* and *ndisreq* are the connection and disconnection signals for the network layer, *tcrsignal* is the transport connection request PDU, *tcasignal* the transport connection accept PDU, *tccsignal* the transport connection clear (connection reject) PDU and *tdatindsignal* is the ASP carrying the reassembled transport data PDU up to the session layer.

The first part of the test purpose in figure 10, shows the network connection phase (*nconreq*, *nconcnf*). The open part is carried out between *nconcnf* and *tdatindsignal*, where a maximum of three signals (MaxDepth 3) may be generated. Next, a *tdatindsignal* must be generated. *ndisreq* is the postamble of the tested system, as well as the default. The goal for the postamble and the default is to drive the system under test to the initial state.

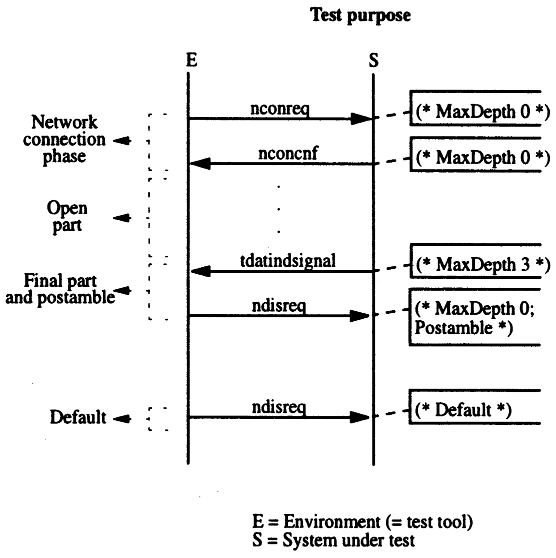

Figure 10 Test purpose for TP0.

Test Case Dynamic Behaviour

Test Case Name: tp0ed_0000
Group: tp0ed_tpexample
Purpose:
Default: tp0ed_default_000
Comments:

Nr	BehaviourDescription	ConstraintsRef	V.
1	START Global		
2	npco ! nconreq	nconreq_c0	
3	npco ? nconcnf	nconcnf_c0	
4	tpco ! tcrsignal (tcr.sref:= tcrsignal.sref)	tcrsignal_c0	
5	(tca.dref:=tcr.sref, auxtca:=tca.dref, tcc.dref:=tcr.sref, auxtcc:=tcc.dref)		
6	tpco ? tcasignal	tcasignal_c0(auxtca)	
7	tpco ! tdtsignal (tdt.em:= tdtsignal.em)	tdtsignal_c0	
8	(eot:=(tdt.em='80'O))		
9	[eot = TRUE]		
10	auxmsg:=Print("spco tdatindsignal")		
11	npco ! ndisreq	ndisreq_c0	P
12	tpco ? tccsignal	tccsignal_c0(auxtcc)	
13	npco ! ndisreq	ndisreq_c0	I

Default Dynamic Behaviour

Default Name: tp0ed_default_000
Group:
Objective:
Comments:

Nr	BehaviourDescription	ConstraintsRef	V.
1	? TIMEOUT Global		
2	npco ! ndisreq	ndisreq_c0	I
3	? TIMEOUT		
4	npco ! ndisreq	ndisreq_c0	F
5	npco ? OTHERWISE		
6	npco ! ndisreq	ndisreq_c0	F
7	tpco ? OTHERWISE		
8	npco ! ndisreq	ndisreq_c0	F
9	spco ? OTHERWISE		
10	npco ! ndisreq	ndisreq_c0	F

Figure 11 Example of a generated test case and its corresponding default for TP0.

The test case depicted in figure 11, is generated for the test purpose of figure 10. Changing the test purpose, for example, MaxDepth 3 by MaxDepth 5 would generate three test cases (including the one depicted in the figure). Lines 2 and 3 match the first two events in the test purpose. Lines 4, 6 and 7 are the generated signals between the second and the third event. They are the transport connection phase plus one transport data PDU. Line 10, represents the *tdatindsignal* produced in the system under test. It is dumped as a *Print* message in TTCN because it is not an observable event in the tester. The operator of the testing tool should verify that it has been received in the system under test. Line 11 is the postamble of the test, carrying the system under test to the initial state, in order to run several test cases in sequence. Line 12 is at the same level of line 6, i.e., the test generation tool does also generate correct alternative behaviour lines. This line means that it is correct behaviour for the system under test to reject the transport connection request issued in line 4. Inconclusive verdicts are assigned to this type of branches, because they state correct behaviour, but they do not fit the test purpose. The default in figure 11 comprises every event not included in the test case, leading the execution of the test to a fail verdict.

Data types are automatically generated from the information included in the specification of the system (and the external data library if needed). Constraints without associated predicates are automatically generated also. In the example, all the constraints, except *tdtsignal_c0*, are automatically generated; *tdtsignal_c0* has an associated predicate (gathered while simulating), i.e., tdtsignal_c0.em = '80'O. The user has to provide a value for the *em* field, that fits the predicate. In this case the system of equations to solve is very simple. Figure 12 illustrates a PDU type definition and a constraint declaration.

PDU Type Definition		
PDU Name: tdtsignal **PCO Type:** SAP **Comments:**		
Field Name	**Field Type**	**Comments**
li	OCTETSTRING[1]	
code	OCTETSTRING[1]	
em	OCTETSTRING[1]	
t_user_data	OCTETSTRING[1..2048]	

PDU Constraint Declaration		
Constraint Name: tdtsignal_c0 **PDU Type:** tdtsignal **Derivation Path:** **Comments:**		
Field Name	**Field Type**	**Comments**
li	'02'O	
code	'02'O	
em	'80'O	
t_user_data	'0123456789ABCDEF'O	

Figure 12 Example of PDU type and constraint.

Coverage measures are automatically computed by the tool. The figures achieved with the test purpose depicted in figure 10 are:

- Signal coverage (black box): 80% (8/10)
- Signal coverage (white box): 65% (11/17)
- State coverage: 86% (6/7)
- Branch coverage: 32% (8/25)

5 CONCLUSIONS

Up to date there is not a final and complete answer for the test automatic generation problem. The GAP tool provides a global and practical solution, easing the test specification, by means of automatizing the process as much as possible in order to obtain optimum test cases reducing costs and time. GAP is embedded in the HARPO toolkit, forming a complete, modular, flexible and upgradeable environment, useful for test suite derivation, validation, execution and maintenance using SDL, MSCs and TTCN.

The defined architecture provides automatic support for test suite generation, which impacts directly on the specification process productivity:

- reducing the testing specification time.
- generating many more test cases than in a manual process.
- the correctness of the test cases is ensured by their automatic nature (derived from the reference system specification).
- better quality test suites (coverage measures).
- being included in HARPO reduces the final testing tool development time.

At the moment of writing this paper, the GAP tool is in its final development phase. Several specifications such as INRES, Transport Protocol TP0 and the Call Diversion Service are being used to test the tool. Since there is no definitive solution to the automatic test generation problem, the intermediate results obtained with a prototype of the tool let us be confident on the fact that GAP is on the right way to achieve its final goal: generate a complete, compilable, TTCN test suite (behaviour, data types and constraints) with realistic (executable against an implementation under test) test cases.

6 REFERENCES

[1] CCITT. Specification and Description Language (SDL). Rec. Z.100. The International Telegraph and Telephone Consultative Committee, 1988. Revised 1993 (SDL-92).

[2] ITU-T. SDL Combined with ASN.1 (SDL/ASN.1). Rec Z.105. The International Telecommunications Union, 1995.

[3] CCITT. Message Sequence Charts (MSC). Rec. Z.120. The International Telegraph and Telephone Consultative Committee, 1992.

[4] ITU-T. Extensions of Basic Concepts for Z.120. The International Telecommunications Union, Draft, December 1994.

[5] ISO. Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1). IS-8824. International Standards Organization, 1987. (See CCITT X.208).

[6] E. Algaba, C. F. Cano, J. I. Sanz, O. Valcárcel. HARPO: Herramientas Automáticas para la Realización de Pruebas OSI. Comunicaciones de

Telefónica I+D, June 1994.

[7] ISO. Information Processing Systems - Open Systems Interconnection - OSI Conformance Testing Methodology and Framework. IS-9646. International Standards Organization, 1989.

[8] ISO. Information Processing Systems - Open Systems Interconnection - OSI Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation. IS-9646-3. International Standards Organization, December 1991.

[9] A. Ek, J. Ellsberger, A. Wiles. Experiences with computer aided test suite generation. Proceedings of IWPTS'93, Pau, September 1993.

[10] J. R. Horgan, S. London, M. R. Lyu. Achieving Software Quality with Testing Coverage Measures. Computer, 27(9):60-69, September 1994.

[11] A. R. Cavalli, B. Chin. Testing Methods for SDLSystems. Proceedings of the Seventh SDL Forum, Oslo, Norway, September 1995.

7 BIOGRAPHY

Esteban Pérez joined Telefónica I+D (R&D Labs.) testing engineering group in 1993 and has been engaged with test generation techniques and in automatizing the development and operation of protocol testing tools.

Enrique Algaba is the project manager of testing engineering group in Telefónica I+D (R&D Labs.). Since joining Telefónica I+D in 1988, he has been engaged in the research and development of protocol testing tools and automatizing the testing process.

Miguel Monedero joined Telefónica I+D (R&D Labs.) testing engineering group in 1995. He has been working since then in the fields of test generation and testing automatization.