INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An experiment in automatic generation of test suites for protocols with verification technology

Jean-Claude Fernandez, Claude Jard, Thierry Jéron , César Viho

N ° 0000

Mai 96

THÈME 1

ISSN 0249-6399

INRIA
RENNES

An experiment in automatic generation of test suites for protocols with verification technology

Jean-Claude Fernandez, Claude Jard, Thierry Jéron , César Viho

Thème 1 — Réseaux et systèmes
Projets Spectre, Pampa

Rapport de recherche n ° 0000 — Mai 96 — 22 pages

Abstract: In this report we describe an experiment in automatic generation of test suites for protocol testing. We report the results gained with generation of test suites based on advanced verification techniques applied to a real industrial protocol. In this experiment, several tools have been used : the commercial tool GEODE (VERILOG) was used for the generation of finite state graph models from SDL specifications, the tool Aldebaran of the CADP toolbox for the minimization of transition systems, and a prototype named TGV (for Test Generation using Verification techniques) for the generation of test suites which has been developed in the CADP toolbox. TGV is based on verification techniques such as synchronous product and on-the-fly verification. These tools have been applied to an industrial protocol, the DREX protocol. The comparison of produced test suites with hand written test suites proves the relevance of the used techniques.

Key-words: protocol, black box testing, test generation, verification, conformance, specification, implementation, test purpose

(Résumé : tsvp)

Jean-Claude Fernandez, Inria/Spectre, Vérimag, Miniparc Zirst, rue Lavoisier, F-38330 Montbonnot Saint-Martin, France. Jean-Claude.Fernandez@imag.fr.

Claude Jard, Thierry Jéron, César Viho, respectively CNRS, Inria and University, IRISA/Pampa, Campus de Beaulieu, F-35042 Rennes, France. (Claude.Jard, Thierry.Jeron, Gagnon.Viho)@irisa.fr

This work has been partially supported by an industrial contract with VERILOG in a study for the DGA (Direction Générale pour l'Armement)

This paper will appear in a Special Issue of Science of Computer Programming, Industrially Relevant Applications of Formal Analysis Techniques, Jan Friso Groote editor, Elsevier Science publisher

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Une expérience de génération automatique de suites de tests pour les protocoles à l'aide de la technologie de la vérification

Résumé : Dans ce rapport nous décrivons une expérience de génération automatique de suites de tests pour les protocoles. Nous énonçons les résultats obtenus par la génération de suites de tests fondée sur la technologie de la vérification appliquée à un protocol industriel. Dans cette expérience, plusieurs outils ont été utilisés: l'outil commercial GEODE (VERILOG) a servi à la génération du graphe d'état fini à partir de spécifications en LDS, l'outil Aldebaran de l'a boîte à outil CADP a permis la minimisation de systèmes de transitions, et un prototype appelé TGV (pour *Test Generation using Verification techniques*) et développé dans la boîte à outil CADP a permis la génération de suites de tests. TGV est fondé sur des techniques de vérifications telles que le calcul de produit synchrone et la vérification à la volée. Ces outils ont été appliqués à un protocole industriel, le DREX. La comparaison des tests produits avec des tests écrits manuellement montrent l'intérêt des techniques utilisées.

Mots-clé : protocole, test boîte noire, génération de tests, vérification, conformité, spécification, implantation, objectif de test

1 Introduction

Testing is a crucial point in software development, and especially for the development of protocols, considering the intrinsic difficulty of their design. Several kinds of tests are used in this context. Among these, conformance testing is a protocol testing method which consists in checking an implementation of a single protocol entity against its specification [ISO92]. Interoperability testing is situated at the service level. It consists in checking that the implementation of several communicating entities satisfies the specification of the global service. In both cases one has a specification and an implementation of a system. Testing consists in checking that the interactions of the implementation with its environment through points of control and observation (PCO) conform with the specification. The implementation under test (IUT for short) is stimulated by a tester which simulates the environment. According to this experiment, the tester produces a *verdict*. One experiment is described by a test case. It is a tree in which each branch is a sequence of interactions decorated with verdicts. Each test case checks a particular property generally defined with a test purpose. Test cases are organized in a hierarchy which constitutes a test suite.

The goal of testing is to find errors in the implementation. Proving correctness is elusive as it is generally impossible to describe a finite set of interactions which could prove the conformance of the implementation with respect to the specification. One of the main problems is then to describe in a systematic way a test suite large enough to have a good confidence in the implementation.

In this paper, we argue on the use of verification techniques for the automatic generation of test suites. We focus on a particular experiment of our generation tool. A more complete description of models and algorithms on which this tool is based is available in [FJJV96]. From the methodological point of view, we have tried to take into account the methods used by practitioners. This is the reason why we use test purposes to select test cases. Informal test purposes are formalized by automata. A similar approach appears in [GHN93], in which test purposes are formalized by Message Sequence Charts (MSC). We will see in paragraph 4.1 what are the main differences between these two approaches. The other input of the tool is a formal description of the protocol in the SDL language [CCI88] which is supposed to be correct i.e. it is the reference model of the protocol. The algorithm which constitutes the heart of the generation tool is our main contribution. It is based on verification technology. Its principle is to traverse a synchronous product of the state graph of the specification and the automaton describing a test purpose, while synthesizing a *test graph*. This test graph is then unfolded in a tree in the TTCN format (Tree and Tabular Combined Notation) [ISO92].

The article is organized as follows. In section 2, we make a brief survey of existing automatic methods for the generation of test suites and sketch the links between test generation and verification. In section 3, we describe the industrial context of the study and give a short description of some tools also used in the same study and some tools used in our experiment. Section 4 describes the ingredients involved in the test generation process, i.e. test purposes, test cases, conformance relation, the specification and test architecture of the DREX protocol. We then describe in section 5 how test cases are generated. Section 6 analyses the results obtained in the experiment. Finally we draw some conclusions in section 7.

2 State of the art

2.1 Automated generation of test suites

Over the last twenty years, a considerable amount of work has been done on the subject of automated generation of test suites according to the economic challenge it reflects. Nevertheless, their use in the real world is still in infancy [LL95]. Beyond the performance of the algorithms, we must elaborate further on their integration in assisted methodologies.

Some automatic methods in protocol testing come from circuit testing and are based on automata theory (see for example [Gon70, Cho78, SD88, VCI89, FBK*91, CKP93]). Their principle is to consider the specification and the implementation as Mealy machines i.e. finite state automata which transitions are labelled with inputs and outputs. The general principle is to test each transition of the specification i.e. reach the source state, apply the input and check that the output is correct and check the target state. They essentially differ on the way the target state is checked. The applicability of these methods requires some assumptions on the specification which are generally too strong to be realistic. Moreover the algorithms are generally too complex and produce very long test cases. Among these methods, the simplest, least powerful but most applicable method is called *transition tour*. Its principle to make a tour of all transitions of the specification without checking the target state.

Some other methods come from testing theory [DNH84, Abr87, Bri88, DAV93, Tre92]. The formalism used to model the implementation, the specification and test suites is labelled transition systems i.e. states and transitions labelled with actions between states. A *conformance relation* between the specification and the implementation [Bri88, BALT90] defines which implementations are correct. The generation algorithm builds a tester i.e. a process which may generate all possible sequences allowing to decide the conformance of the implementation with respect to the specification. Except in [Tre95, Tre96], conformance relations make no distinction between inputs of the IUT which are controllable by the environment, and outputs, which are only observable by the environment. Another problem is test selection as the tester has infinite behaviour but the testing activity must be finite.

As far as we know, there does not exist commercial tools which generate test suites and none of these methods, except transition tours, is really used in the industry of protocols. Practitioners still write test suites by hand or with the help of a simulator. One reason is that all automatic methods are based on a formal specification of the protocol and the specification is often informal. Moreover, practitioners know by their experience that it is not reasonable to try to validate all possible behaviours of their protocol. It is why they use test purposes to select some parts of the protocol that they want to test.

2.2 Verification and test

Verification consists in checking that a specification satisfies a property which may be given by a temporal logic formula or another more abstract specification. Although the verification problem is different from the test problem the models used in both activities are very similar. The operational semantics of specifications can be defined in terms of transition systems. The algorithmic of verification presents a very large spectrum of features: partial verification, on-the-fly checking, reductions, etc, which are all relevant for the problem of test generation from a formal specification. Since we work on verification techniques for several years and we think that verification techniques can be used in the context of test generation, we have decided to extend the open verification toolbox CADP (the Caesar-Aldebaran-Distribution-Package from Vérimag [FGM*92]) with a test generation feature.

3 Context of our experiment

3.1 Industrial context

The experiment related here has been conducted during an industrial contract for the *Direction Générale pour l'Armement* of the French Army. Partners of this contract are VERILOG, CNET (Centre National d'Etude des Télécommunications), Cap Sesa Régions and our two research groups (Spectre from Grenoble and Pampa from Rennes). This study started in November 1994 and ended in November 1995. The goal was to prove that the automatic generation of test sequences is feasible and profit-earning in an industrial context. Three tools have been studied and/or developed, TVéda-V3

(CNET), Topic (VERILOG) and our prototype TGV (INRIA). In order to compare the methods and tools, these three tools had to generate test suites starting from the same SDL specification of the DREX protocol (see paragraph 4.3) and test purposes in natural language. It appears that finally the consortium agrees on the different components of a realistic test generator, and that TGV represents a good demonstrator of the main ideas. This is why we think that TGV and the results obtained with it deserves to be presented. TVéda and Topic are briefly described below. The paper will focus on the results of TGV.

3.2 Description of some tools

In this section, we briefly describe the main principles of TVéda and Topic V2, the two other test generation tools involved in the study. We then describe the tools and environment used in our experiment.

3.2.1 TVéda

Tvéda-V3 is a test generation tool from CNET [Pha94] which accepts both Estelle and SDL formal description techniques and can produce test suites in TTCN format. Tvéda-V3 may compute test kernels with two strategies: symbolic computation or reachability analysis. Only symbolic computation was used in the study. The principle is to avoid the enumeration of data values of the specification. The specification is translated into an extended finite state machine (EFSM). TVéda computes a path from a source state of the EFSM to a target state in accordance with conditions on variables values. This path is transformed into a tree when non-determinism of the specification is considered.

3.2.2 Topic V2

Topic V2 is a prototype of VERILOG developed inside the simulator of the commercial tool GEODE (see below) and based on a previous prototype Topic [MRE94]. Its principles are quite similar to those of [GHN93]. It takes as inputs an SDL specification and test purposes described as MSCs. The algorithm computes a graph constrained by the test purpose and unfolds it into a tree which is translated into TTCN.

3.2.3 Tools used by TGV

GEODE [ALHH93] is an SDL commercial tool developed by VERILOG. This tool supports requirement analysis, graphical design for data, architecture, communication and state machines, simulation and code generation. The simulator allows interactive simulation or automatic simulation for exhaustive verification.

GEODE has been used by TGV as an SDL front-end in order to produce the state graph of the specification (see the details in section 5).

CADP, the Caesar Aldebaran Distribution Package [FGM*92] is a toolbox developed by Verimag at Grenoble. It is composed of several tools and environments among which Aldebaran and Open/Caesar that were used by TGV:

Aldebaran is a tool which performs reduction and comparison of graphs according to various equivalence relations and preorders.

Open/Caesar is an open environment for rapid prototyping of verification algorithms or other algorithms based on traversals of transition systems. This is allowed by the presence of libraries for the management of graphs and memory. Transition systems may be given implicitly by functions produced by the Lotos compiler Caesar or by other compilers or explicitly in the Aldebaran

graph format. The implicit representation is interesting for "on-the-fly" algorithms i.e. algorithms based on traversals of the transition system without explicit construction of the complete transition system.

TGV has been developed in the CADP toolbox. It uses the libraries of Open/Caesar for the management of graphs and memory. In the experiment, it used an explicit representation of transition systems but an on-the-fly version is currently being developed. Aldebaran has been used for the minimization of state graphs produced by GEODE and translated into the Aldebaran graph format.

4 Ingredients of the test generation process

In this section we define test purposes and how they have been formalized, the conformance relation used, test cases and their verdicts. We then present the specification of the DREX protocol used in the study and its test architecture.

4.1 Test purposes and their formalization

Experts in test know that they cannot test all parts of the IUT. It is the reason why they use test purposes in order to select some particular parts to be tested. A test purpose defines a property of the system that one particularly wants to check on the IUT. It is generally composed of two parts:

- a context which describes a constraint that must be applied to the IUT before testing.
- a behaviour which defines the sequencing of some interactions between the IUT and its environment.

In our experiment with the DREX protocol, the context part is described by filters used as inputs to the GEODE simulator. A filter describes a set of forbidden actions. It is particularly useful for fixing the results of some undefined operators of the SDL specification.

The behavioural part of the test purpose is specified by an automaton. The transitions of the automaton are labelled with interactions of the specification with the environment. The automaton is acyclic except implicit loops in each state. This allows to describe test purposes at an abstract level without considering all interactions of a sequence. The automaton has accepting states labelled with the keyword *Accept*. A sequence of observable actions applied to this automaton is accepted if it has a sub-sequence labelled with actions of the automaton reaching an accepting state.

In an approach by [GHN93], test purposes are formalized by Message Sequence Charts (MSCs). An MSC describes an observable behaviour which cannot be interrupted i.e. observable interactions are forced to be immediate successors. Thus no abstraction is possible and an MSC describes the part of the test case which is the most difficult to compute. However, a mixing approach could be investigated. An automaton or any other formalism could have some transition sequences which are interruptible and others which are not.

4.2 Test cases and conformance

A test case is a tree in which each branch describes a sequence of interactions between the tester and the IUT. In protocol testing, test cases are often described with the Tree and Tabular Combined Notation (TTCN [ISO92]). The role of a test case is to detect if an IUT conforms with its specification according to a particular test purpose.

This implies to formally define what is conformance. The conformance notion used in this experiment is basically the relation $ioconf_{\mathcal{F}}$ described in [Tre95, Tre96]. Informally, an IUT I conforms with a specification S, according to a set of traces $\mathcal{F}$ if, after all observable trace in $\mathcal{F}$, the outputs of

I are included in the outputs of S. In our case, $\mathcal{F}$ is the subset of traces of S which are accepted by the automaton of the test purpose.

In order to detect non conformance, some transitions of test cases are decorated with verdicts. The meaning of verdicts is the following.

FAIL means that the IUT does not conform to the specification. According to the conformance relation, a FAIL verdict is assigned to each input of the tester which does not correspond to any output of the specification. This is achieved by adding an implicit Otherwise FAIL in each state. FAIL verdicts are also assigned to timeouts (see below).

(PASS) means that the sequence from the initial state corresponds to an interaction sequence of the specification and is accepted by the test purpose automaton. It is a temporary verdict because after this sequence another sequence called *postamble* should reach the initial state. The execution of this postamble can still produce FAIL verdicts.

PASS is a definitive verdict meaning that the initial state has been reached after a (PASS) verdict. The sequence between (PASS) and PASS identifies a postamble.

INCONCLUSIVE is used when an input of the tester corresponds to an output of the specification which either cannot lead to a (PASS) verdict or leads to a behaviour that is not considered in the test case because testing cannot be exhaustive.

4.3 The DREX protocol and its specification

The protocol used for our experiment is the DREX protocol. It is a military protocol which allows the access to the transit network of the French Air Force, defined in the framework of Integrated Service Military Network. It is quite similar of the DM protocol which is the analogous of the D protocol in the civil framework. The DREX protocol runs on a network called SOCRATE and connects several MTBX (Telecommunication Means of Air-Bases) at the T interface (see Figure 1).

Figure 1: Reference configuration of the DREX protocol.

This protocol has been chosen for several reasons. The protocol had to be an industrial one in order to prove the feasibility of automatic test generation methods on realistic specifications. An SDL specification of a similar protocol, the DM protocol, was still available, written by Cap Sesa Régions, partner of the project. Test cases written by hand by Cap Sesa with the help of the GEODE simulator were available. They have served as a basis for comparison with automatically generated test suites.

The specification of the DREX protocol used in the study contains a functional description of the offered services and the specification of the user interface for the call command. It models the service of the DREX protocol on the entire network SOCRATE.

Only a subset of services offered by the DREX protocol have been specified in SDL. These services are Priority, Roving User, Call Forwarding, Implicit Partitioning of Users, Safety Path and User to User Signaling. A generic SDL specification has been written and instantiated for each service. So, there were as many SDL specifications as services. This means that tests have been produced for each

service separately but the interaction between services has not been studied. The SDL specification models only one connection between two MTBX. They are called DR for the requesting MTBX and DE for the requested MTBX. The size of the SDL specification corresponding to each service is about 2000 lines.

4.4 Specification, test purpose and consistency relation

A test purpose denotes an important part of a specification which should be tested. As the specification and the test purpose are formalized (in the context of automatic generation of test suites), we must also formalize the relation between the formal specification and the formal test purpose. This is done by means of a preoder relation between two automata, one for the test purpose the other for the specification. We call this relation the *consistency relation*. This is a weak notion of satisfaction meaning that at least one sequence of the specification is accepted by the automaton of the test purpose. A test case is derived from the specification and the test purpose if and only if both agree with the consistency relation.

4.5 Test architecture

In our experiment with the DREX protocol, we were only concerned with tests of the six services described above. There was three kinds of tests to generate. Tests of the subscription record of the MTBX, end-to-end tests which check the transmission of information between two MTBX and service tests which check the service given by the DREX protocol between two MTBX. As most tests concern the service provided by DREX, we can say that we are interested in interoperability testing.

The test architecture chosen is shown in Figure 2.

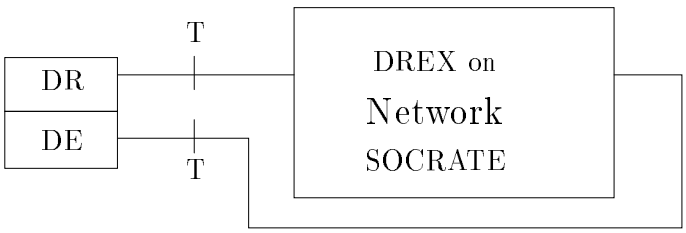

Figure 2: Test configuration for DREX.

The implementation of the DREX protocol on the entire network is seen as a black box. The tester models the behaviour of two MTBX, named DR and DE connected to the network by two PCOs at interface T. The communication between an MTBX and the network is supposed to be asynchronous. According to [ISO92], we are in the context of a simplified Multi Party Testing with no lower tester but two synchronized upper testers behaving as MTBX users.

The test architecture of the DREX has an influence on the generation of test cases. We are faced to the case where the tester cannot interact directly with the IUT but interacts with a test environment in which is placed the IUT. This has some influence on the way the tester can control and observe the IUT. As the communication between the tester and the IUT is asynchronous and two PCOs are considered, this may introduce concurrency in the behaviour. Suppose for example that the protocol receives the ETAB message from DR (the requesting MTBX). It treats the message with an internal procedure trt_etab and replies by sending in sequence ETAB to DE (the requested MTBX) and APP_COURS to DR. In an external view the receptions of ETAB in DE and APP_COURS in DR are concurrent (see Figure 3 for an illustration).

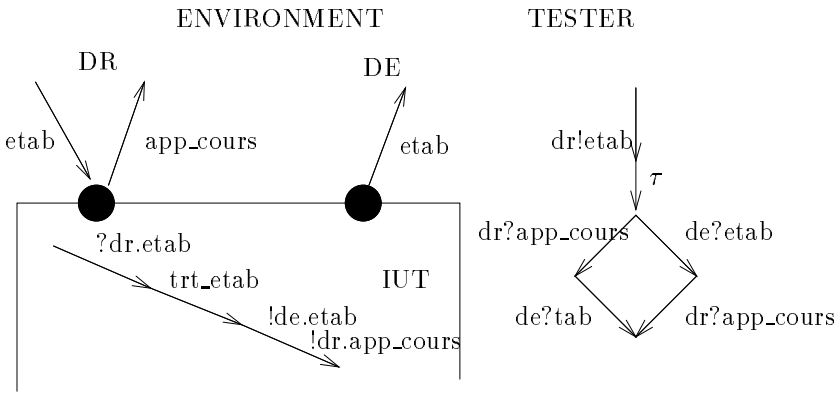

Figure 3: Concurrency in testing.

We have considered the influence of the test environment in the following way. We first assume the *reasonable environment hypothesis* which says that, each time the environment sends a message to the network, it waits until stabilization. This means that no new message can be sent by the environment until it receives all specified outputs of the protocol. This hypothesis allows to avoid concurrency between inputs and outputs by limiting the crossings of messages.

Asynchronous communication is treated by considering that the tester does not check the IUT directly but checks the IUT in its test environment. Thus we transform the specification by coupling it with a model of the test environment. This can be seen as an extension of the approach of [VTKB93]. In their paper, they only considered one bidirectional PCO while we have to cope with two PCOs. This was possible because of the reasonable environment hypothesis and because of the particular form of the specification. In this specification, an SDL transition is always an input followed by internal actions and outputs, and there does not exist output loops. Thus we only had to transform the sequence of outputs of the transition into diamonds modeling concurrency.

5 Generation method

In this section we describe the TGV package. We detail each step of the test generation, describing inputs and outputs of each software component. We then describe more deeply the algorithm which constitutes the kernel of the TGV package.

5.1 The TGV package

The programmes developed for this study are intended to be coded in a complete integrated tool in which one could specify a protocol, analyze it with different verification methods and generate test suites. We have particularly taken care of the fact that all our algorithms can work on-the-fly during the computation of the specification graph.

However, our ambition in this experiment was only to prove the feasibility of the approach. Thus the TGV prototype has been developed outside the verification tool GEODE (see Figure 4). As a consequence, the TGV package (in grey in figure 4) takes as input a state graph produced by GEODE. The second input of TGV is the automaton formalizing the behavioural part of the test purpose. A test purpose may specify one or more test cases. In the context of our experiment, we have choosen to select arbitrarily one test case for one test purpose. The TGV package outputs one test case in the TTCN standard language, distinguishing the control (behaviour) and data (constraints) parts. As usual, the intermediate form MP can be graphically displayed using TTCN-GR.

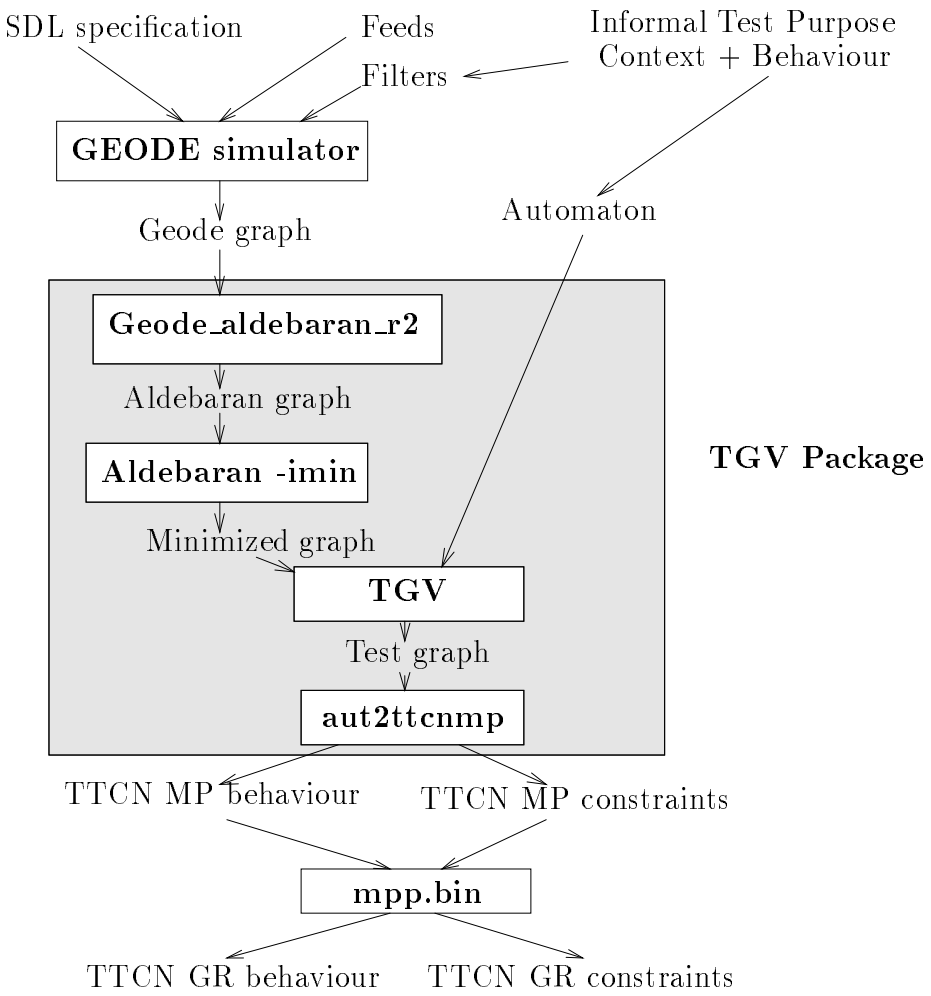

Figure 4: Architectural view of the TGV package.

TGV has been developed in the toolbox CADP described above. This was particularly useful because CADP is an open environment which does not depend on a particular language. We have used GEODE as a SDL front-end but our prototype could be used with other languages like Lotos or Estelle. We now detail each part of the generation of a test case.

5.2 Overview of the generation method

The generation of a test case can be decomposed in several functional parts which are performed by different tools (see Figure 4):

Step 1 : State graph generation with GEODE

The inputs of GEODE are:

- an SDL specification.
- a set of messages (feeds) that the environment can send to the implementation. This is used to close the SDL specification with a restricted environment in order to limit the size of the state graph generated by GEODE. Message parameters can be fixed or not. Feeds are supposed to describe messages accepted by the implementation.

- a set of filters which specify some forbidden transitions. These filters represent the constraint part of the test purpose.

The GEODE simulator is used in verification mode with the *reasonable environment hypothesis*. It generates the state graph corresponding to all possible behaviours of the protocol closed by feeds and constrained with filters.

Step 2 : Translation and mirror image

This step is performed by a tool called geode_aldebaran_r2. The graph generated by GEODE is translated into a graph in the Aldebaran format which represents the observable behaviour of the specification in its test environment.

As the implementation is seen as a black box, we are only interested in the observable behaviour of the protocol i.e. interactions with its environment. Thus internal actions of the specification are replaced by an undistinguished invisible action denoted by τ. The test environment is also taken into account (see paragraph 4.5). This means that the graph is transformed by the introduction of diamonds corresponding to interleavings which model concurrency. They are built on-the-fly during a traversal of the graph.

As we now consider the tester view, we also perform a mirror image of the graph i.e. inputs are replaced by outputs and vice versa.

Another feature of geode_aldebaran_r2 is filtering of message parameters. This helps in producing readable test suites in which only parameters of the message which are significant for a particular test purpose are preserved.

Step 3 : Minimization and determinization with Aldebaran

At this step Aldebaran minimizes the graph produced by the preceding step with respect to the τ^*a equivalence and determinizes the result.

The τ^*a equivalence preserves safety properties and the minimization with respect to this equivalence is very efficient in practice. Minimization roughly means that only observable actions are preserved and states with same observable behaviours are collapsed. Collapsing is in fact not necessary, the important thing here is only the hiding of τ actions. This is crucial in an on-the-fly perspective because on-the-fly minimization is not efficient. During this step, the graph is also determinized because the tester cannot see non-deterministic choices of the protocol. The graph produced is called **external view graph** and represents the external view of the protocol behaviour.

Step 4 : Test graph generation

This is the kernel of the tool. The algorithm is based on a depth-first traversal of a synchronous product of the external view graph and the automaton representing the test purpose. As usual, this traversal can be limited by a maximal depth. States of the product are composed of a state p^{TP} of the automaton and a state p^{S} of the graph. A transition t is fireable in a state $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ of the product and leads to $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ in two cases. Either t is fireable in the graph in state p^{S} and leads to q^{S} and t is fireable in the automaton in state p^{TP} and leads to q^{TP}; or t is fireable in p^{S} and leads to q^{S} but t is not fireable in p^{TP}, in which case $p^{\mathrm{TP}} = q^{\mathrm{TP}}$. This means that the specification always progress but the automaton progresses only when it is synchronized with the specification.

During the traversal, several computations are performed:

- the algorithm checks the consistency relation between the test purpose and the specification. This is done during the descent in the traversal. When an accepting state is reached, a postamble is computed by a search of a shortest path to the initial state.

- a **skeleton graph** is synthesized while backtracking. This graph contains some sequences without loop of the synchronized product which contain an accepting state of the automaton and reaches the initial state of the specification (postambles are added). The graph satisfies the *controlability condition* which says that the tester controls its outputs. Thus, if an output is fireable in a state of the graph, no other transition is fireable.
- the transitions of the skeleton graph are decorated with verdicts with the meaning described in paragraph 4.2. This is illustrated by Figure 5.

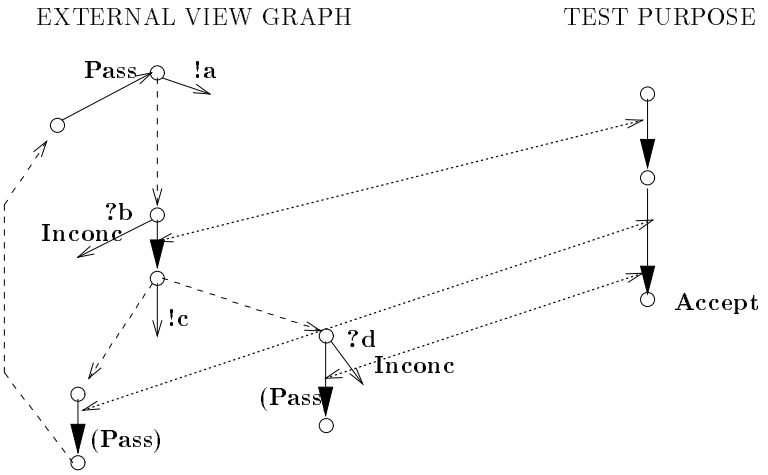

Figure 5: Construction of the test graph.

This algorithm is a depth first search of the product graph combined with a breadth first search for the computation of the shortest postambles. Its complexity is linear in the size of the state graph of the specification times the size of the test purpose automaton. The algorithm is explained in more detail in the annex and in [FJJV96].

Step 5 : Management of timers Timers are used to detect deadlocks or unobservable loops of the IUT. When an input is expected by the tester, it does not want to wait for an unbounded time because an unobservable error may have occurred in the IUT. The difficulty in managing timers comes from the necessary distinction between concurrency and choice. Notice that this occurs only for inputs. Timers are managed by TGV in the following way (see Figure 6 for an example).

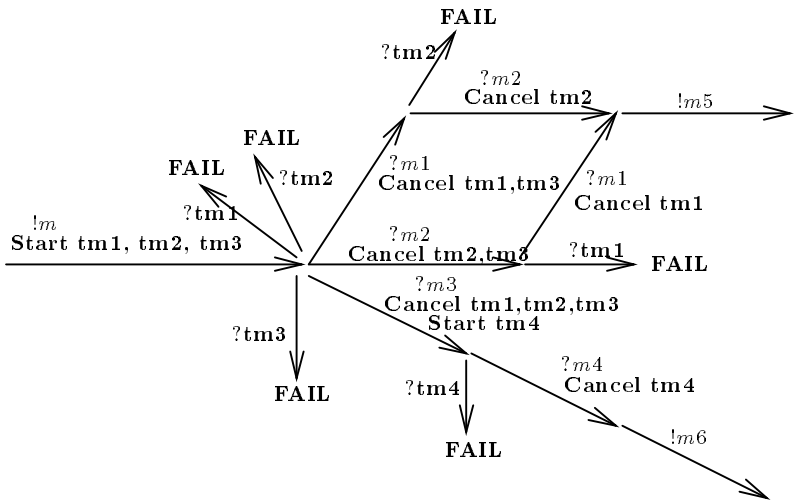

Figure 6: Management of timers in TGV

A timer **tm** is associated to each possible input of the message **m** by the tester. The timer **tm** is started in the last transition which necessarily precedes the input of **m**. In all states where an input of **m** is expected, a timeout of **tm** may occur. When **m** is received, each timers **tm'** corresponding to an expected input **m'** which is not concurrent with **m** must be cancelled. Thus **tm** is at least cancelled when **m** is received but also in each other input choice not concurrent with **m**.

Timers are generated during a depth first traversal of the skeleton graph. Concurrency between inputs is computed during the traversal. The rules for the management of timers are original and much more complete than those found in the literature which never consider the concurrency problem. These rules insure that all deadlocks of the IUT are detected by a timeout.

At the end of this step, we have a graph called **test graph** containing all the informations needed for the generation of a test case in TTCN. This graph is in the Aldebaran format. All informations such as message names, message parameters, verdicts, timers are contained in transitions labels.

Step 6 : TTCN generation

In this last step, the tool aut2ttcnmp takes as input the test graph generated at the preceding step, and produces two files. The first file describes constraints, in fact the parameters of messages. The second file describes the behavioural part of the test case obtained by unfolding the test graph in a tree. An identifier is used to refer to the parameters of messages in the other file. Both files are in TTCN MP format. They are then translated into files in TTCN GR format by **mpp.bin**, a tool of VERILOG.

6 Experiment with the DREX protocol

In this section we describe how the experiment has been carried out on the DREX protocol. We first describe how test purposes have been formalized to serve as inputs to the test generation. We then give the quantitative results of the experiment and finally qualitative results by comparison with hand written test suites.

6.1 Formalization of test purposes

Test purposes were given in natural language and had to be formalized. An informal test purpose describes the context of the test and an abstract property of some interactions.

The context of the test is concerned with properties of the subscription record of the users or properties of the network. The subscription record is modeled in the specification by the use of undefined operators on abstract data types, generally boolean operators involving choices between transitions of the specification. An example of context is "*The requesting MTBX does not have access to this service*" or "*The network is saturated*". We have formalized these contexts by filters, a mechanism of the GEODE simulator which allows to disable some transitions.

The remainder of the test purpose describes sequencing properties between inputs and outputs of the tester and network, as well as properties on parameters of messages. For example, "*The requested MTBX receives an ESTABLISHMENT message containing the parameter User to User Information (IUU). It sends an ALERT message with parameter IUU. Verify that the network sends to the requesting MTBX an ALERT message without parameter IUU*".

As we are mainly concerned with the sequencing of actions, the relations between parameters have been treated in a simple way. We only consider equality between parameters. As test cases describe the behaviour of the testing process, the network actions have to be translated into tester actions. In our example "the network sends to the requesting MTBX" has to be replaced by an input of the MTBX. Care must be taken in this transformation because if the test purpose describes two output actions of the network through different PCO, then this may involve concurrency in the reception of these messages, translated into interleaving. This behavioural part of the test purposes has been translated into automata in the Aldebaran format (see below the automaton of the example).

```
des(0,3,4)
(0,"de?etab(fuu = (if_present = false, value = fuu_b), iuu = (if_present = true, value = iuu_b))",1)
(1,"de!alert(fuu = (if_present = false, value = fuu_b), iuu = (if_present = true, value = iuu_b))",2)
(2,"dr?alert(fuu = (if_present = false, value = fuu_b), iuu = (if_present = false, value = iuu_b))",3)
Accept 3
```

In general, test purposes were as simple as this example. Only a small subset of test purposes contains interleavings of actions.

6.2 Quantitative results

Generation of the protocol graph

The experiment has been carried out on a SparcStation 5 with 32MB of RAM.

The size of each specification (one for each service treated as described in section 4.3) is approximately 2000 lines of SDL. Depending on the environment fixed by the feed mechanism and the test context fixed by filters, the size of the state graph generated by GEODE was from 1000 to 10000 states and from 1500 to 15000 transitions. Once minimized the size of external view graphs was roughly divided by 100. The time spent by GEODE for the generation of these state graphs and Aldebaran for the minimization and determinization was between 3.5 s and 427 s. The memory size (data + stack) used by GEODE and Aldebaran was between 788 k-bytes and 18748 k-bytes. This is the most expensive part of the test generation process which is often called "state space explosion". Our limits are those of verification tools. With exhaustive verification tools like GEODE we can reasonably treat specifications having up to one million states. However, these limits can be pushed with symbolic exploration (state space implicitly represented with Binary Decision Diagrams for example) or on-the-fly generation (generation of tests while traversing the state space).

Formal description of test purposes

We have considered 40 test purposes. The size of an automaton specifying a test purpose is from 2 to 6 transitions and 3 to 5 states. The mean time spent for the formalization of one test purpose has been about 5 minutes. This includes the description of the context by filters and the specification of the behaviour part by an automaton.

Generation of test suites

The time spent by the TGV package for the production of one test case (including step 2 to step 5, starting with the graph produced by GEODE) was between 0.7 s to 1.7 s and the memory size used by TGV was between 228 k-bytes to 408 k-bytes. These good results can be explained by the linear complexity of the main algorithm. The size of test cases produced varies between 2 transitions and 100 transitions.

6.3 Comparison with hand written tests cases

A test suite of 54 test cases corresponding to the 54 informal test purposes had already been written by hand. These test cases had been written without any formal definition of conformance. But the examination of the specification and test cases showed us that the conformance relation was almost the same as $ioconf_{\mathcal{F}}$ (see paragraph 4.2). Hand written test cases have served as a basis for the analysis of our results. We have compared test cases generated automatically with hand written ones and have found a lot of similarities but also a lot of differences. The differences observed are principally due to the fact that TGV treats concurrency and timers in a systematic way.

Before to sketch the main differences, let us make some remarks. TGV is just a prototype in which we have put much emphasis on the behavioural part of test cases. So it could not take into account all the possibilities of the TTCN language. We produce pseudo-TTCN in which some fields of TTCN, like test group, test purpose, comments and default libraries are not generated. A test case generated by TGV contains a preamble and postambles attached to the test body, although TTCN allows to separate them in different files. TGV does not consider the hierarchical structure of a test suite as it is intended to be integrated in a complete tool.

The reasonable environment hypothesis

In our opinion, the most important differences concern the behaviour part of test cases. A lot of differences can be explained by the hypothesis made on the behaviour of the environment. We have assumed the *reasonable environment hypothesis* because we thought that it corresponds to the behaviour of the tester. But in hand written test cases, this hypothesis is not always assumed. The reason is probably because test developpers already have in mind the test run process on the real implementation and implicitly use their experience with the protocol.

This is the case in the following example. The test purpose says that after DR!ETAB, we must have DE?ETAB. In the specification, there is an input of ETAB from DR followed by the outputs of ETAB to DE and APP_COURS to DR. Asynchronism implies the concurrency between DE?ETAB and DR?APP_COURS. The test case generated by TGV is in table 1 while the hand written test case is shown in table 2.

The difference between the two test cases lies between lines 8-12 of the automatic test case and lines 7-8 of the hand written test case. In the latter, after DE?ETAB, the action DR?APP_COURS is not considered. It is supposed that the tester can decide to make outputs before inputs (control has priority on observation). Nevertheless, we think that it is safer to assume the reasonable environment hypothesis and wait for DR?APP_COURS before applying the postamble. If one wants to link test cases, it may happen that forgotten receptions like DR?APP_COURS have an influence on a subsequent test case which may lead to a FAIL verdict although a PASS verdict should be produced.

Nr	Behaviour Description	Constraints Ref	Verdict
1	dr! etab, START tetab, START tapp_cours	etab0	
2	dr? app_cours, CANCEL tapp_cours		
3	de? etab, CANCEL tetab	etab0	(PASS)
4	dr! flib, START tflib	flib1	
5	de? flib, CANCEL tflib	flib1	PASS
6	? tflib		FAIL
7	? tetab		FAIL
8	de? etab, CANCEL tetab	etab0	(PASS)
9	dr? app_cours, CANCEL tapp_cours		
10	dr! flib, START tflib	flib1	
11	de? flib, CANCEL tflib	flib1	PASS
12	? tflib		FAIL
13	? tapp_cours		FAIL
14	? tapp_cours		FAIL
15	? tetab		FAIL

Table 1: Test case produced by TGV

Nr	Behaviour Description	Constraints Ref	Verdict
1	dr! etab, START TW303, START TWAIT	etab0	
2	dr? app_cours, CANCEL TW303		
3	de? etab, CANCEL TWAIT	etab0	(PASS)
4	+ POST_LIB_DRS_DES		
5	? TIMEOUT TWAIT		FAIL
6	dr! flib	flib1	
7	de? etab, CANCEL TWAIT, CANCEL TW303	etab0	(PASS)
8	+POST_LIB_DRS_DES		
9	? TIMEOUT TW303		FAIL
10	dr!flib, CANCEL TWAIT	flib1	FAIL
11	? TIMEOUT TWAIT		FAIL
12	dr!flib, CANCEL TW303	flib1	FAIL

Table 2: Hand written test case (details discarded). The postamble POST_LIB_DRS_DES cancels timers and performs the outputs DR!FLIB and DE!FLIB.

Postambles

Another difference which also appears in the previous example concerns postambles. In hand written test cases there are postambles after FAIL verdicts. In our point of view, if a FAIL verdict has been produced, it is impossible to know in which state the implementation is. Thus the effect of a postamble starting from the current state of the IUT is not foreseeable. Thus it is impossible to insure that the initial state is reachable without a reset. However, postambles after INCONCLUSIVE verdicts are relevant. This is not yet implemented in TGV but poses no problem.

Postambles are often not the same as in hand written tests. This is due to the choice of messages sent by the tester to the implementation and specified in feeds.

Concurrency

Concurrency between events are sometimes forgotten in hand written test cases. It often happens in complex test cases because it is difficult to foresee the behaviour of the protocol without an automatic tool, especially in the case of concurrency. The dramatic consequence is that some hand written test cases are not correct: their execution might produce a FAIL verdict for a conformant implementation.

Timers

In hand written test cases, timers corresponding to internal protocol timers are used. This is not conformant to the black box testing paradigm. Internal timers are not observable and should not be used in test cases. They can be used only in order to help in adjusting the values of timers when test cases are executed. But they are not sufficient as transmission delays also have to be taken into account. We have considered another point of view in which timers used in test cases are events of the environment.

Specification constraints

TGV often produces INCONCLUSIVE verdicts which do not appear in hand written tests. This typically occurs when we have not sufficiently constrained the context by filters. Thus the state graph has more behaviour that in the case where the context is correctly fixed. As choices are made by the protocol, these supplementary behaviours start with receptions of the tester. But as these behaviours do not lead to a (PASS) verdict, the receptions have to be considered as INCONCLUSIVE. Thus these differences between automatic and hand written test cases have not been considered as errors. However, for each test case it is necessary to exactly specify which context is considered. Otherwise, FAIL verdicts could be pronounced in place of INCONCLUSIVE verdicts.

Interpretation of informal test purposes

Differences due to the interpretation of test purposes is unavoidable because of the ambiguity of natural language. Without a deep knowledge of the protocol, it is sometimes impossible to know what is the exact meaning of a test purpose. This is a strong argument for the use of formalized test purposes by automata or other formalisms like MSCs or temporal logics. Ambiguities disappear with formalization and everybody can agree on the exact meaning of test purposes.

6.4 Comparison with test cases produced by TVéda and Topic V2

The main advantage of TGV with regard to the other tools used in the experiment is the fact that TGV was the only tool able to take into account the asynchronism between the tester and the IUT. Another advantage of TGV is the management of timers which is more elaborated.

The comparison with Topic V2 was relatively easy because they used almost the same principles. The algorithms of TGV are more efficient and complete than those of Topic V2. But Topic V2 allows to specify test purposes in MSCs or extended automata, produces declarations and constraints and has a better user interface.

TVéda is more complete than TGV as it produces complete TTCN descriptions with declarations and constraints, test purposes, comments, etc. Comparison between TGV and TVéda concerning the behavioural part of test cases is more difficult as their principles are very different. In TVéda, test purposes are implicit (each transition of the SDL specification is a test purpose). Thus TVéda produced more test cases. Some of them did not correspond to any informal test purpose, thus could not be produced by TGV or Topic. Some others were discarded afterwards because they had inconsistent constraints on message parameters. A new version of TVéda based on reachability analysis was still in development during the study [CGPT95]. Comparing our results with results produced by this new version would be interesting.

7 Conclusion

During one year, we have carried out a study which goal was to prove that formal methods can be successfully applied for the generation of test suites. The interest of our industrial partners was

motivated by economical considerations. They want to reduce the time spent for the development of test suites by hand.

We have shown that some formal techniques developed in the area of verification could be useful and profit-earning for the automatic generation of test suites. We have formalized the notion of test preorder and have developed a generation algorithm based on a depth traversal of a synchronous product of the graph representing the external behaviour of the protocol specification and an automaton formalizing a test purpose. A prototype has been developed which accepts as inputs specifications written in different languages (at present connections with SDL and Lotos are available). We have used standardized languages like SDL for the specification and TTCN for test suites and the best known algorithmic tools have been applied. We have been able to generate test suites for the DREX protocol which is a real industrial example, without coming up against a considerable algorithmic complexity. Graphs have been generated and translated in a reasonable time. All this augurs well of the viability of a forthcoming integrated tool.

Produced results conform with what was intended. We have been able to automatically generate test cases corresponding to all hand written ones. Moreover, the gain in quality is undeniable as some hand written tests cases have been proved incorrect while they were correctly generated with TGV. This proves, if it was necessary, that automation is indispensable for test suite generation. Particularly the systematic treatment of concurrency and timers offers more safety. The only shortcoming is that our automatic method requires time to specify the protocol and formalize test purposes. But this is amply balanced by the benefit in time and quality of automatic methods. This is particularly true in a long term perspective where formalization will be more integrated in the development process.

Like all automatic test suite generation methods, we suppose that a protocol specification is given. But we also need formalized test purposes. Thus, from a user point of view, the effort is in the specification of the protocol and the conception and formalization of test purposes. We think that this conception of test purposes can be done during the process of specification. Afterwards according to the nature of test purposes, they can be modelled using automata, MSCs or temporal logic.

If we consider the type of algorithm used, we can envisage to develop these methods on-the-fly. In fact, the CADP environment gives access to high level primitives for graph traversal allowing to couple simulation and verification, or test sequence generation. The advantage is that one can use the algorithm on specifications which state graph cannot be completely generated. The drawback is that we cannot minimize the state graph with Aldebaran but this is not absolutely necessary.

For on-the-fly generation of test cases, CADP must also treat data in a more complete way. This is not yet done as graphs in the Aldebaran format and the transition function of CADP do not give access to data types. Offering this possibility would allow to really apply on-the-fly verification techniques for the generation of test suites and would also enrich verification methods.

Acknowledgments

We wish to thank the referees for helpful comments on the paper and our partners in the experiment related here, in particular J.-P. Ropars from Celar, B. Algayres and L. Doldi from VERILOG, S. Lebricquir and N. Texier from Cap Sesa Régions, M. Phalippou from Cnet and L. Nedelka from Irisa.

References

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3), 1987.

[ALHH93] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON Project: A VALidatiON environment for SDL/MSC Descriptions. In *SDL 93 Forum*, 1993.

[BALT90] E. Brinksma, R. Alderden, J. Langerak, R. Van de Lagemaat, and J. Tretmans. A Formal Approach to Conformance Testing. In J. De Meer, L. Mackert, and W. Effelsberg, editors, *Second International Workshop on Protocol Test Systems*, pages 349–363, North Holland, 1990.

[Bri88] E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification VIII*, pages 63–74, North-Holland, 1988.

[CCI88] CCITT/SGx/WP3-1, Specification and Description Language, SDL. *CCITT Recommendation Z.100*, 1988.

[CGPT95] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In *8th Int. Workshop on Protocols Test Systems*, Evry - FRANCE, September 1995.

[Cho78] T. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3), may 1978.

[CKP93] A.R. Cavalli, S.U. Kim, and Maigron P. Automated Protocol Conformance Test Generation Based on Formal Methods for LOTOS Specifications. In Elsevier Science Publishers B.V., editor, *Protocol Test Systems V*, pages 212–222, Montréal, 1993.

[DAV93] K. Drira, P. Azéma, and F. Vernadat. Refusal graphs for conformance tester generation and simlification: a computational framework. In *Protocol Specification, Testing and Verification XIII*, IFIP Transactions, North-Holland, 1993.

[DNH84] R. De Nicola and M. Henessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[FBK*91] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6), june 1991.

[FGM*92] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Tool Box for the Verification of Lotos Programs. In *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. *To appear in Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, July 1996.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In Elsevier Science B.V. (North-Holland), editor, *6th SDL Forum*, pages 253–266, Færgemand, O. and Sarma, A., Darmstadt (Germany), 1993.

[Gon70] G. Gonenc. A method for the design of fault-detection experiments. *IEEE Transactions on Computing*, C-19, june 1970.

[ISO92] 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). *International Standard ISO/IEC 9646-1/2/3*, 1992.

[LL95] R. Lai and W. Leung. Industrial and academic protocol testing : the gap and the means of convergence. *Computer Networks and ISDN Systems*, 27:537–547, 1995.

[MRE94] J. Montiel, R. Roth, and Donaldson A.J.M. (Eds). *Methods for QoS Verification and Protocol Conformance Testing in IBC - Results and Further Recommendations.* Race Project R2088 TOPIC, Deliverable 15, DocR2088/DAT/TMS/DS/P/015/b1, November 1994.

[Pha94] M. Phalippou. Test sequence using Estelle or SDL structure information. In *FORTE'94*, Berne, October 1994.

[SD88] K Sabnani and A. Dahbura. A protocol testing procedure. *Computer Network and ISDN Systems*, 15(4), 1988.

[Tre92] J. Tretmans. *A formal approach to conformance testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[Tre95] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In *8th Int. Workshop on Protocols Test Systems*, Evry - FRANCE, September 1995.

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs and Quiescence. In T. Margaria and B. Steffen, editors, *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'96, Passau, Germany*, March 1996.

[VCI89] S. Vuong, W. Chan, and M. Ito. The UIOv method for protocol test sequence generation. In *Second International Workshop on Protocol Test Systems*, Berlin, October 1989.

[VTKB93] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asychronous testing. In G. Von Bochman, R. Dssouli, and A. Das, editors, *Fifth International Workshop on Protocol Test Systems*, IFIP Transactions, North Holland, 1993.

Annex : the generation algorithm of TGV

The generation algorithm combines four main functions described below. But for the sake of readability, we only present the first part of the test generation algorithm:

- It checks the consistency relation and synthesizes the skeleton graph. The algorithm performs a depth-first traversal of the graph associated to the synchronous product between the external view graph and the automaton specifying the test purpose. It checks that a sequence in the synchronous product reaches a state in which the automaton component is an accepting state. In this case, it searches a sequence in the external view graph leading to the initial state of the graph. Meanwhile, starting from accepting states, it synthesizes the skeleton graph which
- It computes the shortest postamble, using a breadth first search. In the complete algorithm this search is called when an accepting state is reached.
- It ensures the *controlability condition* of the tester on its outputs. If an output is enable in a state of the test graph, then all the other transitions are discarded. This is done during the traversal and allows to cut some parts of the graph.
- The skeleton graph is decorated with verdicts and timers to produce the test graph used to generate TTCN suites. Verdicts are assigned to transitions while backtracking in the traversal. Timers are produced afterwards during another depth first search.

To allow the DFS computation, several data structures and functions are required:

Data structures:

- Γ is a stack with elements $\gamma = (\alpha, (q^{\mathrm{TP}}, q^{\mathrm{S}}), l) = (\gamma.\mathrm{Act}, \gamma.\mathrm{State}, \gamma.\mathrm{Succ})$ where α is the last action preceding $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ (ϵ for initial state $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$), $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ is a couple of states of the synchronous product, l is the list of successors of $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ (see below). The stack Γ is managed through the usual operations "push", "pop" and "top".
- *Result* is a boolean value, initially set to *false*, and set to *true* if the test purpose is consistent with the specification. Its value is returned by *dfs*.
- Q is the set of states of the synchronous product. $V \subseteq Q$ is the set of stored visited states which are no longer in Γ.
- Pre, Test and Post are the sets of nodes in the preamble, the test body or the postamble respectively. They are updated by the function create_trans.
- *Nodes* is a subset of V whose elements are the vertices of the test graph. Initially, *Nodes* is `Accept` $\times\{q^{\mathrm{S}}_{\mathrm{init}}\}$. *Trans* is the set of transitions of the test graph.

Functions:

- The function *succ_list* delivers the successors list of a given compound state. *choose_and_remove* chooses a transition and removes it from the list.
- *create_trans* updates transitions and nodes of the test graph.

---

function dfs $((q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$: state) **returns** boolean **is**
begin
  $V := \emptyset$; $\Gamma := \emptyset$; *Result := false* ; Post := `Accept` $\times\{q^{\mathrm{S}}_{\mathrm{init}}\}$; Pre := $\emptyset$; Test := $\emptyset$;
  Nodes := Post; Trans := $\emptyset$; Pass := $\emptyset$; (Pass):=$\emptyset$; Inconc:=$\emptyset$;
  l := succ_list$((q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}))$; push $((\epsilon, (q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}), l), \Gamma)$;
  while $(\Gamma \neq \emptyset)$ **loop**
    $(a, (p^{\mathrm{TP}}, p^{\mathrm{S}}), l)$:= top (Γ) ;
    if $(l \neq \emptyset)$ **then**
      choose_and_remove $(b, (q^{\mathrm{TP}}, q^{\mathrm{S}}), l)$;
      if $((q^{\mathrm{TP}}, q^{\mathrm{S}}) \notin (V \cup \Gamma \cup$ `Accept` $\times \{q^{\mathrm{S}}_{\mathrm{init}}\}))$ **then** /* new state */
        l':=succ_list$((q^{\mathrm{TP}}, q^{\mathrm{S}}))$; push $((b, (q^{\mathrm{TP}}, q^{\mathrm{S}}), l'), \Gamma)$;
      else if $((q^{\mathrm{TP}}, q^{\mathrm{S}}) \in$ Nodes) **then**
        create_trans$((p^{\mathrm{TP}}, p^{\mathrm{S}}), b, (q^{\mathrm{TP}}, q^{\mathrm{S}}))$
      end if
    else /* $l = \emptyset$ and $(a, (p^{\mathrm{TP}}, p^{\mathrm{S}}), l)$ = top(Γ) */
      pop(Γ)
      if ($(p^{\mathrm{TP}}, p^{\mathrm{S}}) \in$ Nodes and $\Gamma \neq \emptyset$) **then**
        create_trans(top(Γ).State, a, $(p^{\mathrm{TP}}, p^{\mathrm{S}})$);
      end if
      V := V $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$;
    end if
  end loop
  if $((q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}) \in$ Pre and (`Accept` $\times \{q^{\mathrm{S}}_{\mathrm{init}}\} \subseteq V)$) **then** *Result* := *true*; **end if**
  return(*Result*)
end

---

procedure create_trans$((p^{\mathrm{TP}}, p^{\mathrm{S}}), b, (q^{\mathrm{TP}}, q^{\mathrm{S}}))$ **is**
begin
  Nodes := Nodes $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
  Trans := Trans $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}}), b, (q^{\mathrm{TP}}, q^{\mathrm{S}})\}$
  if $((q^{\mathrm{TP}}, q^{\mathrm{S}}) \in$ Post) **then**
    if $((p^{\mathrm{TP}} = q^{\mathrm{TP}}))$ **then** Post := Post $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
    else Test := Test $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
    end if
  else if $((q^{\mathrm{TP}}, q^{\mathrm{S}}) \in$ Test) **then**
    if $((p^{\mathrm{TP}} = q^{\mathrm{TP}}_{\mathrm{init}})$ and $(q^{\mathrm{TP}} = q^{\mathrm{TP}}_{\mathrm{init}}))$ **then** Pre := Pre $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
    else Test := Test $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
    end if
  else if $((q^{\mathrm{TP}}, q^{\mathrm{S}}) \in$ Pre) **then** Pre := Pre $\cup\{(p^{\mathrm{TP}}, p^{\mathrm{S}})\}$
  end if

end

---

function succ_list $((p^{\mathrm{TP}}, p^{\mathrm{S}}))$ **returns** list of (action, state) **is**
begin
  if $(((p^{\mathrm{TP}} = q^{\mathrm{TP}}_{\mathrm{init}})))$ **then**
    return$(\{(\alpha, (q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}})) \mid p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}}\})$
  else
    return$(\{(\alpha, (q^{\mathrm{TP}}, q^{\mathrm{S}})) \mid p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}}$ and $(p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}} q^{\mathrm{TP}}$ or $(\neg(p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}})$ and $q^{\mathrm{TP}} = p^{\mathrm{TP}}))\})$
  end

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399