Verification and test generation for the SSCOP protocol[*]

Marius Bozga[†], Jean-Claude Fernandez[†], Lucian Ghirvu[†], Claude Jard[‡], Thierry Jéron[‡], Alain Kerbrat[§], Pierre Morel[‡], Laurent Mounier[†]

March 9, 1999

Abstract

Many formal tools are now efficient enough to deal with small to medium size systems. Working with larger systems requires not so much to improve these tools, but to use them in combination, applying one tool for what it is most efficient for, and using its results to improve the applicability of the other tools. This paper presents such a combination, illustrated on an industrial protocol, large enough to break any brute force approach. Two research teams allied their forces with a software engineering tools maker in order to analyze, verify and generate automatically tests for this protocol, by the extension and the interconnection of their various tools. The results obtained give some hints on a methodology for the formal validation of large systems.

Keyword: ATM, Protocol, SSCOP, Static Analysis, Verification, Model-checking, Conformance Testing, Test Generation.

[*]This work has been done in the context of FORMA. FORMA is a national project funded by the French army, the CNRS and the ministry of research.

[†]VERIMAG-Centre Equation, 2 rue de Vignate, 38610 Gieres, France. This work was partially supported by Région Rhône-Alpes, France. E-mail:{Marius.Bozga,Jean-Claude.Fernandez, Lucian.Ghirvu, Laurent.Mounier}@imag.fr, Tel: 33+ 4 76 63 48 53, Fax: 33+ 4 76 63 48 50

[‡]IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, E-mail: {Claude.Jard, Thierry.Jeron, Pierre.Morel}@irisa.fr, Tel: 33+ 2 99 84 74 64, Fax: 33+ 2 99 84 71 71

[§]VERILOG, 150 rue Nicolas Vauquelin, 31106 Toulouse Cedex , E-mail:kerbrat@verilog.fr

1 Introduction

The Sscop protocol (Service Specific Connection Oriented Protocol) is an industrial protocol, part of the ATM stack (Asynchronous Transfer Mode), presently standardized by the Itu-T [22].

The deployment of this protocol in telecommunications networks raises the following questions :

- consolidation of the protocol specifications: does it correctly ensure the service requested in all the possible configurations?
- design of correct and powerful tests to detect the nonconformity of implementations with respect to the standardized specification.

It is clear that the first point is essential in an objective of broad dissemination of the protocol. Standardization is not a sufficient guarantee of correctness. Sscop is a complicated object which superimposes many protocol mechanisms concerning complex situations from the point of view of memory and time management. Design bugs could resist the primarily manual work of the experts. At least the conditions of guaranteed correct operation are not all completely clarified.

The economic stake of the second point is also significant: only certified implementations should be disseminated. The quality of the certification depends on the quality of the tests.

There are actually several test suites available (that one can buy for a few tens of thousands of dollars) which deserve to be improved:

- to guarantee that a conformant implementation will not be rejected (it is a difficult problem in an asynchronous testing architecture where it is necessary to foresee the phenomena of concurrency on different interfaces and collisions of the stimuli and the observations),
- to let slip through only a reduced number of non-conformant implementations (the tests must be as complete as possible).

This situation led the Cnet (the research center of France Telecom) to start an activity of formal verification and automatic test generation on the Sscop. The entry point was the Sdl (Specification and Description Language) description provided as part of the Q 2110 [22] document. To evaluate the capacity of industrial and academic tools to check properties and to generate full-scale tests, the Cnet subjected the Sscop as industrial case study to the Forma project.

Forma is a French national action supported by the direction of the army, the Cnrs and the ministry of research. It aims at the evaluation and the transfer of techniques of formal validation of temporal specifications. It is structured in well targeted operations gathering research and industrial teams around a case study and short term objectives (2 years). The Sscop experiment rallied four research teams at CEA (Saclay), LSV (ENS Cachan), Irisa (Rennes) and

VERIMAG (Grenoble) in cooperation with CNET (Lannion) and the software company VERILOG (Toulouse).

Our article presents the results obtained at the end of the first year and the work achieved on SSCOP in the context of SDL specifications and tools.

The plan of the article is the following. We start by presenting the SSCOP protocol and its formal specification then we present the tools used. The results are gathered in three topics: preliminary analysis of the formal specification, the verification of communicating entities and the automatic generation of conformance tests. We try to present the perspectives from both the point of view of academic research and industrial results.

2 The SSCOP protocol and its specification

2.1 The SSCOP protocol

The SSCOP protocol is standardized under reference ITU-T Q2110 [22]. Originally, it was conceived to reliably transfer data between two high bandwidth network entities. Although its design makes it ready to treat significant volumes of data, currently its use is confined in the indication layer of the ATM. However, it is reasonable to think that it will be employed to transfer high volumes of data in future applications. SSCOP is one of the underlayers of the layer AAL (ATM Adaptation Layer). The main role of AAL is to adapt the service provided by the ATM physical layer to the type of data passing by connections established between two ends.

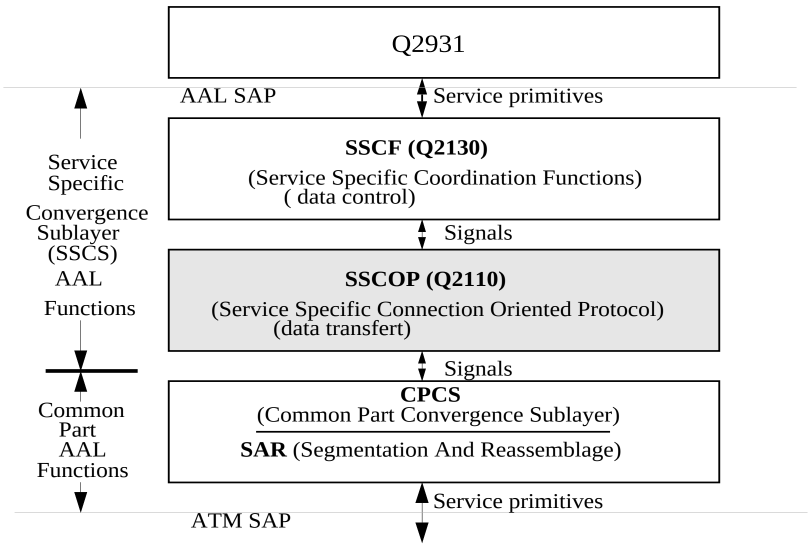
Q2931
AAL SAP
Service primitives
Service Specific Convergence Sublayer (SSCS) AAL Functions
SSCF (Q2130)
(Service Specific Coordination Functions)
(data control)
Signals
SSCOP (Q2110)
(Service Specific Connection Oriented Protocol)
(data transfert)
Signals
Common Part AAL Functions
CPCS
(Common Part Convergence Sublayer)
SAR (Segmentation And Reassemblage)
Service primitives
ATM SAP

Figure 1: Situation of SSCOP in the ATM stack

2.1.1 Provided services

Sscop provides to the upper layer (Q2931 for example) the following services:

- Sequencing: SDUs (Service Data Units) submitted by the upper layer are numbered in the order in which they will be submitted for transfer,
- Protocol error detection and recovery: the receiver detects loss of PDUs (Protocol Data Units) and asks for selective retransmission,
- Flow control: achieved by a classical window mechanism with size determined by the receiver,
- Error reporting to the management layer,
- Keep connection alive in the case of a long absence of data transfer,
- Local data retrieval: if necessary, local data can be retrieved among not yet released SDUs,
- Connection control: establishment, release and resynchronization,
- Data transfer with two modes: guaranteed or not,
- Window state indication.

2.1.2 Exchanged signals

Sscop exchanges two types of signals with its environment:

- signals exchanged with the upper layer SSCF (Service Specific Convergence Function) defined in [23]. These signals are internal primitives of the ATM stack (ASP for Abstract Service Primitives) and are generally not observable from outside. Their coding is not standardized.
- signals exchanged with a peer Sscop entity. These signals are PDUs and are accessible by a tester. Their coding is standardized.

2.2 The Sdl specification of the Sscop protocol

The Sscop standardization document [22] contains an informal description of the Sscop and an Sdl description of the protocol. This Sdl description has been coded by Cnet using the Sdl editor ObjectGéode (Verilog). It consists in approximately 2000 lines of Sdl described by one single process. The specification is open in the sense that the environment is not described. This description is centered on signaling and some simplifications have been made according to Sscop implementations available in Cnet. Some other simplifications, such as removal of unobservable internal actions, have also been done by the Cnet in order to adapt it to the purpose of test generation.

This specification has been simulated in CNET using the SDL simulator OBJECTGÉODE. The CNET also applied its test generator TVÉDA to derive abstract test cases. These test cases have been translated to executable test cases and applied to real implementations. This work is described in [9]. We will come back on the produced test suites in section 6.

3 Tools

3.1 OBJECTGÉODE (VERILOG)

OBJECTGÉODE is a real time systems development toolset, supporting the use of three formalisms:

SDL is the Specification and Description Language, standardized by the Z.100 recommendation [24]. SDL is the main language of the toolset, it allows to describe the architecture and the behaviour of a real time distributed system.

MSC is the Message Sequence Charts language. It is standardized by the Z.120 recommendation [25]. It is usually combined with SDL, as it allows to describe runs of the system, with a more or less abstract view of its architecture.

OMT is the Object Modeling Technique defined by J.Rumbaugh and al [26]. Within OBJECTGÉODE, it is mainly used to describe data.

OBJECTGÉODE includes graphical editors and compilers for each language. It provides also a C code generator and a simulator which allows some debugging and some verification of SDL programs. Finally, a test generation prototype is also included.

As the focus of this work is on verification and test generation, we will present in more details the simulator tool, which was necessary in several phases of the verification and test generation works. The test generation prototype was not applied, as its functionalities are largely covered by the tool TGV (see section 3.2).

The *Simulator* tool

The *Simulator* allows to simulate runs of the system, without having to actually execute it in a real environment. It can be seen as a sort of abstract debugger, as it allows to simulate the description step by step, to undo execution steps, to set break points and to watch the contents of variables and queues. Finally, it also allows to record, visualize as MSCs or replay some simulation sequences. It is also more than a debugger, as it allows to perform automatic simulation, either randomly or exhaustively, with systematic comparison of the behavior with special state machines called observers.

The simulator working principle is based on the model checking principle. The GSMCOMP SDL compiler produces the needed functions for the graph generation and some of the data structures for the model's representation. The *Simulator* itself provides the data structures for the model exploration (hashtables, stacks and heap management). It integrates exploration programs such as deadlock and livelock search, assertion checking and comparison with observers. All the functions and the data structures provided by the simulator are accessible via a well-defined API.

Observers: The core of the verification methods of OBJECTGÉODE is based on the observers [4]. They can be directly written using the GOAL language or compiled from MSCs. Observers are state machines which are executed side by side with the SDL description. Every time an event occurs (for example the firing of the whole transition, or the input of a signal, or an informal decision), the observer checks if it is an event it is able to recognize (there is a transition from its current state which matches the system's transition). If it is the case, it executes its corresponding transition, otherwise it ignores the event. The states of observers can be qualified either as success or error states. During the comparison of a description with an observer, sequences leading to error states can be saved as diagnostics. Moreover, observers can be considered as a substitute to the user for exhaustive and random simulation modes. An observer uses a set of *probes*, given as access paths to the entities (blocks, processes, queues, variables) to be observed. These probes allow to observe events like transition firing, communications of signals, creation or stopping of processes, time progression or procedure calls. They also give the possibility to change the program behavior, by changing the value of variables, so they can be used for example for fault-injection in the system.

3.2 TGV (IRISA-VERIMAG)

TGV is a prototype tool developed by our two teams in Rennes and Grenoble [11, 12, 18]. Its aim is to automatically generate test cases for conformance testing of distributed systems, starting from a formal specification of the system and test purposes allowing to select test cases. These test cases are composed of interaction sequences. An interaction is either an output of the tester which is proposed to the implementation, or an input which is an expected answer of the implementation according to its specification. Test cases also contain timers which ensure the finiteness of the test execution and verdicts which are produced according to the conformance or not of the implementation with respect to the specification. The conformance relation relating implementations to specifications is allmost identical to the **ioco** relation of Tremans et al [27]. Unformally it says that an implementation conforms to its specification if after any observable trace existing in the specification, outputs produced by the implementation are foreseen in the specification and the implementation may block only if the specification also allows it.

3.2.1 Main principles

The principle of TGV is to compute a test case from a specification of the system and a test purpose. The algorithms are not described in detail here but these algorithms ensure that produced test cases are unbiased in the sense that any implementation which conforms to its specification will not be rejected by a test case produced by TGV.

The specification must be given in a language which operational semantics allows to represent its set of possible behaviours by a state graph. This state graph is either explicit or implicit leading to two different modes of using TGV: *explicit* and *on-the-fly* generation.

Explicit generation: In this case the state graph of the specification is previously computed by a simulation tool. The test generation then necessitates several phases. The first step is to translate the state graph into a format accepted by TGV. Then, as testing considers traces of observable interactions, the internal actions are abstracted (τ^*-reduction), and the state graph is determinized and then minimized [11]. The resulting graph represents the observable behaviour of the specification on which the main algorithm of TGV can be applied. The main drawback of this approach is the state explosion problem which limits the applicability of this method to small specifications. An alternative is to generate test cases on-the-fly as described above.

On-the-fly generation: TGV can also be applied to implicit state graphs. The principle is to compute a test case while constructing, in a lazy strategy, only the part of the state graph which is necessary for the test case computation. This is called on-the-fly generation. The advantage of this method is to be able to compute a test for large specifications with very large and even infinite state graphs. In order to be applicable, TGV must be linked with an API of a simulation tool which provides some basic functions for the graph construction, namely the function which computes the initial global state, the function which computes fireable transitions, the function which computes the global state reached from a previous global state by firing a transition, and functions which compare global states and store them in memory. From an algorithmic point of view, the difficulty comes from the fact that successive transformations described above for explicit graphs (except minimization) are applied here to implicit graphs during their construction. This imposes that algorithms are conceived using APIs.

Test purposes: A test purpose characterizes an abstract property that the system should have and that one wants to test. In TGV it is used to select a test case from all possible behaviours of the specification. It is formalized by a finite automaton labelled with some interactions of the specification. This automaton has *accepting* states which define the accepted language and *refusal* states which allow to cut the exploration of some parts of the state graph in order to better guide the test case search. The automaton allows some abstraction using wild card transitions. This contrasts in particular with the test generation method used in SAMSTAG [16] which uses test purposes defined as MSCs describing complete sequences.

Main algorithm: TGV is based on algorithms coming from the model-based verification domain. These algorithms check that a specification satisfies a property given by a logic formula or by an automaton. Some of them are based on traversal of the state graph. If the property is not satisfied, a diagnostic sequence can be extracted. The algorithm of TGV adapts this principle for test generation. Searching a sequence of the specification which satisfies the test purpose can be seen as producing a sequence that characterizes the non satisfaction of the negation of this test purpose. In fact, TGV is even more complex as it produces a set of sequences i.e., a sub-graph. Very efficient algorithms exist for doing this, and in particular those which perform on-the-fly verification are well adapted for on-the-fly generation of test cases. The principle is to traverse a synchronous product of the state graph of the specification and the test purpose automaton. Test cases are synthesized while backtracking from reached accepting states.

Testing architecture: Testers often do not directly communicate with implementations. Such situation arises when communications take place through fifo channels in an asynchronous way. This implies phenomenon such as message collisions (the tester sends A and waits B while the implementation sends C) and concurrency on different PCOs[1] (the implementation sends A on PCO1 and B on PCO2 in sequence but the tester may receive A and B in any order). By the way this creates lost of control and observation of the tester on the implementation. In order to treat this correctly and produce correct test cases, the asynchronous communication must be a parameter of the test generation. ISO 9646 recommends to generate a generic test case and then to take into account the test architecture for the production of an abstract test case. But for the case of asynchronous communication, it is easy to prove that this strategy does not work as the production of a generic test case may loose some informations which are necessary in order to derive a correct abstract test case. For simple architectures one could also treat the problem by a transformation of the state graph of the specification. This has to be done on explicit or implicit state graphs. In the second case, this implies to integrate the transformation in the kernel of TGV. This was the first strategy adopted by TGV. But this slightly complicated TGV and the implemented transformations were not complete. So, our strategy is now to describe the test architecture inside the specification and to derive abstract test cases from this new specification. This complicates the specification and produces a supplementary explosion of the state graph. Thus it would be difficult to apply this for explicit generation. But it works quite well with on-the-fly generation. This has been experimented for SSCOP.

[1]PCO: Points of Control and Observation i.e. interfaces through which the implementation is controlled and observed by testers

3.2.2 Languages and companion tools

TGV was first developed in the context of conformance testing of telecommunication protocols. So it is based on standard languages of the domain. Thus it is applicable to specifications written in SDL [24] or LOTOS [1] and can produce test cases in the TTCN language (Tree and Tabular Combined Notation) defined as a part of [2]. Nevertheless, it is relatively independent of any language because it manipulates the standard model of state graphs which is used to represent the possible behaviours of specifications, test purposes and test cases.

On-the-fly generation has been applied successfully in the context of LOTOS specifications using OPEN-CÆSAR [14] from the CÆSAR-ALDÉBARAN toolset of VERIMAG and INRIA Rhones-Alpes [10, 6]. In the context of SDL specifications we have also applied on-the-fly generation using an open version of the OBJECTGÉODE simulator from Verilog [28] which offers an API with state graph construction functions described above [19]. In this case some libraries of CÆSAR-ALDÉBARAN are also used for graph storage.

The output of TGV is a test case which is given by a graph in an ad hoc format. We can translate this test case into TTCN. In the context of telecommunication protocols, it is important to make this translation as TTCN is de facto the standard for writing test cases.

3.3 Other tools from VERIMAG

VERIMAG is developing for 10 years a toolset dedicated to the design and verification of protocols. Some of them are distributed as part of the CÆSAR-ALDÉBARAN toolset [10, 6].

Some tools have been adapted or designed especially for this work, in order to be connected efficiently with the OBJECTGÉODE toolset. These tools can be classified according to their functionality :

Generation of intermediate form: SDL2AUT has been partly developed for this case study. This tool translates an SDL specification into a set of extended automata, one per SDL process. The transitions of these automata are labelled with basic SDL actions (input, output, task,...).

In this form, the protocol became easily tractable by our verification tools such as ALDÉBARAN, MMGGRAPHIC, for the verification of some global and very abstract properties. It was also possible to apply static analysis techniques, which happened to be crucial for the limitation of the state explosion occurring during the full verification and test generation.

Minimization and comparison of behaviours: ALDÉBARAN allows to minimize a state graph, or to compare a state graph with a more abstract one, with respect to equivalence relations preserving the observable behaviour of the system. In particular, ALDÉBARAN uses *simulation* and *bisimulation* relations such as strong and weak bisimulation [21], branching bisimulation [15], and safety equivalence [5].

Evaluation of temporal logic formulas: EVALUATOR provides on-the-fly verification of temporal properties over finite state graphs. The temporal logic considered in its case is the alternating-free μ-calculus [20]. Like many other similar on-the-fly verification tools, EVALUATOR is based on a local resolution method for boolean equation systems [13]. Such systems are usually derived from the state graphs when expressing the semantics of temporal properties. EVALUATOR completes the other available analysis tools, which are essentially based on behavioral verification.

Visualization: MMGGRAPHIC is a tool for visual analysis and diagnosis of distributed systems. It uses a global and abstract view of the system. The tool performs an interactive and visual exploration based on iterative local refinements corresponding to a *zoom* effect on some states of the system's model, i.e. the state graph.

It works as follows : if we minimize the state graph of the system (preserving the behaviour), by considering only a small (e.g. less than 5) subset of observable events, we usually obtain a model small enough to be drawn and analyzed visually. Some parts of this very abstract model can be detailed by extending the set of observable actions and then reiterating this process.

VERIMAG took advantage of the APIs of OBJECTGÉODE, and connected the tools SDL2AUT and EVALUATOR respectively to the SDL compiler and to the simulator [19]. The benefits of such connections are numerous :

- A tool such as SDL2AUT can be designed without having to re-implement a full SDL compiler, yet keeping the upward compatibility with future evolutions of SDL.
- The model checker EVALUATOR can work on-the-fly on SDL specifications, thus avoiding some limitations due to the state explosion problem.

Other translation tools have also been implemented, in order to convert the explicit model produced by the OBJECTGÉODE simulator into a model suitable for ALDÉBARAN, MMGGRAPHIC, and the explicit version of TGV.

4 Static analysis of the SDL specification

Our first attempt to verify the initial specification was to directly generate the state graph using OBJECTGÉODE. But even for very simple scenarii, this task cannot be accomplished, mainly because of the complexity of the data part.

4.1 Abstract behavioural analysis

The following consideration allows to abstract away the variables: when simulating exhaustively an SDL specification without evaluating the values of the variables, we obtain a super set of the program behaviour. Indeed, the guards being not evaluated, each transition of the control flow graph is fireable. Therefore, there exists a *simulation* [21] between the original program and this abstract model. Then, it is possible to check some class of properties on this abstract behaviour, for instance the expected properties of the service. The interest of these verifications is their weak cost, since the abstract graph is much smaller than the original one. In particular, the verifications we performed consisted in comparing this abstract graph with the one supplied by the standard to model the interactions between adjacent layers.

This comparisons with respect to the safety equivalence, was performed with ALDÉBARAN. Some subtle errors, such as omission of timers setting, were found using this method.

The main steps of the analysis are summarized below:

- generation of a reduced model from the SDL specification, with SDL2aut. We obtain a graph with about 1000 states.
- minimization of this model, using ALDÉBARAN, with respect to strong bisimulation. We obtain a graph with about 300 states.
- properties checking on this resulting graph.

However, this first abstraction was too coarse to verify the most interesting properties. So, we now turn back to the original specification in order to perform more sophisticated analyses.

4.2 Preliminary simplifications

When we want to model the behaviour of an SDL specification with a state graph, two parameters have an influence on the size of this model:

- the state number, depending on the size of the variables domains.
- the state vector size, depending on the number of the program variables,

Our model based approach does not allow to perform parameterized verification. Therefore, we choose to restrict the size of the variables domain to the lowest values specified by the standard.

Another simplification was the suppression of useless variables, some of them detected by the OBJECTGÉODE compiler and some others detected by hand, such as for example, some PDUS only relevant for the implementation (reserved records, ...). Moreover, some parts of the specification have been slightly rewritten in order to suppress redundant variables (local variables used in a state to construct a PDU before its emission).

Furthermore, consider the SDL implicit variables **sender** associated with each process. It contains the identification of the process from which the last message was received. This variable may take many values. As a consequence, some states, behavioural equivalent, are distinguished. But this implicit variable is not referenced in this specification. The use of an intrusive observer with

ObjectGéode allowed us to assign an unique value to this variable without changing the behaviour of the specification.

These coarse simplifications may be refined strongly by performing *live variables analysis* [3, 29] of the specification, as explained in the next subsection.

4.3 Live variables analysis

A variable is *live* in a state if there is a path from this state along which its value is used before it is redefined. Otherwise, it is dead. Live variables can be computed by performing a *backward analysis* on the model. We modify slightly the usual definition of Def and Use from [3]. We define $Use(t)$ to be the set of variables that are used in the transition t, and $Def(t)$ to be the set of variables that are defined (assigned) in the transition t.

A variable is *live* on a state p if there is a transition t, such that $p = source(t)$ (the transition source state) and either the variable is live on $target(t)$ (the transition target state) and not in $Def(t)$, or if it is in $Use(t)$.

This information is computed by solving the least fixpoint equations:

$$\forall p \in Q \qquad Live(p) = \bigcup_{\{t \mid source(t)=p\}} (Use(t) \cup (Live(target(t)) \setminus Def(t))$$

An important reduction of the model *state space* can be obtained by taking into account the live variables for each control state. In fact a model state must be strictly characterized by the values of the live variables, not by the values of *all* model variables. Or, in other words, we must not distinguish states differing only on values of dead variables. Thus, we can define a *living equivalence* which is stronger than the strong bisimulation.

The model reduction that we propose consists in directly computing the quotient model $S_{/\sim_{live}}$. This can be done in a straightforward manner at the model generation time using various techniques. For example we can directly use the living equivalence to test equality of newly generated states instead of the strong (complete) equality of state vectors. Another simple way is to modify the initial automaton by introducing systematic *(re)sets* of dead variables to some given value. This optimization has been implemented in a tool especially designed for this case study. In this case study, a spectacular benefit we obtained is the reduction of the state graph size by a division of 200.

4.4 Perspectives

The study of a complex Sdl specification points out the importance of static analysis to optimize the automaton modeling the behaviour. Some other analysis, such as constant or interval propagation, are currently studied. Moreover, the use of the property we want to check (resp. the test purpose used to generate a test case) could improve even further the verification step (resp. the test generation step).

5 Verification of a pair of communicating SSCOP entities

The purpose of the static analysis stage described in the previous section was both to detect most of the coarsest errors or omissions in the original protocol specification, and to abstract it to facilitate its verification by model-checking. Therefore, it now remains to check for its correctness in more details.

However, because of its complexity, and particularly since there does not exist any "exhaustive" reference behaviour of a protocol entity (i.e., valid for *any* environment), it is clear that this correctness cannot be established in the general case. Consequently it is necessary to concentrate our verification effort to a set of representative scenarios, for which specific properties are expected.

More precisely, the system we consider in the following consists in a pair of protocol entities, communicating through bounded fifo channels. Thus, the communication layer is assumed to be reliable and no signal loss is allowed. Moreover, each entity is able to exchange a given set of signals with its upper layer (the SSCF layer). In particular, by restricting to an appropriate set the signal sequences received by each entity from the SSCF layer, it becomes possible, using OBJECTGÉODE, to generate a model of the corresponding protocol behaviour, and, when this model is finite, to verify it with ALDÉBARAN.

In the remaining of the section we detail some scenarios of the verifications that we performed using this approach[2].

5.1 Connection establishment

We considered a first scenario devoted to a connection establishment between two entities. For this scenario, the signals accepted at any time[3] by each entity from its SSCF layer are:

- the request signal for a connection establishment ("AaEstablishRequest");
- the response signal to a connection establishment ("AaEstablishResponse").

The resulting state graph generated by OBJECTGÉODE contained 15 000 states, and was reduced modulo strong bisimulation to 5 000 states using ALDÉBARAN.

For checking the correctness of the connection establishment, we considered the two informal requirements:

Req 1: any connection *request* received by a protocol entity *can* be followed by a connection *confirmation* issued by the same entity;

Req 2: any connection *request* received by a protocol entity is *eventually* followed by a connection *confirmation* issued by the same entity.

[2]A preliminary approach was conducted by the LSV team (under the supervision of A. Finkel) where the connection-disconnection phase was manually translated into Promela for model-checking using SPIN

[3]following the *reasonable feed* simulation policy of OBJECTGÉODE.

These two requirements were formally expressed in the μ-calculus, and evaluated on the protocol state graph using EVALUATOR.

Although the first requirement was clearly verified, the second one happened to be false, and a diagnostic sequence was produced by the tool. The analysis of this sequence showed that the connection establishment may fail due to the expiration of one of the timers ("TimerCC") associated to each entity. This timeout happens when the PDU exchange required by the connection establishment takes too much time. The connection is then aborted, which is correct with respect to the standard. Consequently, **Req 2** was rewritten as follows and verified using EVALUATOR:

> Any connection *request* received by a protocol entity, not followed by a timeout of "TimerCC" occurring on any entity, *is eventually* followed by a connection *confirmation* issued by the same entity.

Therefore, we can conclude that under our assumptions a correct connection establishment is guaranteed by the SDL specification.

5.2 Disconnection

To analyze the protocol behaviour during a disconnection step we now add the disconnection *request* signal ("AaReleaseRequest") to the set of signals received by each entity from the SSCF layer. The resulting state graph generated by OBJECTGÉODE contained now 30 000 states, and it was reduced to 8 000 states by ALDÉBARAN.

The informal requirements we considered were the following:

Req 3: any disconnection *request* received by a protocol entity *is eventually* followed by a disconnection *indication* issued by the other entity;

Req 4: any disconnection *request* received by a protocol entity *can be* followed by a disconnection *confirmation* issued by the same entity;

Req 5: any disconnection *request* received by a protocol entity *is eventually* followed by a disconnection *confirmation* issued by the same entity.

These three requirements were expressed in terms of μ-calculus formulas, and evaluated on the protocol state graph using EVALUATOR, leading to the following results:

- **Req 3** is true, which means that any disconnection request is correctly transmitted from one entity to the other;
- **Req 4** is true, which means that a connection *can* be correctly released by the two entities;
- however **Req 5** happened to be false, and a diagnostic was produced by EVALUATOR.

Here again, the analysis of this diagnostic showed that a disconnection request may not be confirmed, either because the connection has never been correctly established before, or because it has been already released in the meantime. Furthermore, this last situation occurs either because of a timeout (of the "NoResponse" timer), or because the other entity has previously requested for a disconnection. Since these two scenarios do not contradict the SSCOP standard, the disconnection step can be considered as correctly specified by the SDL protocol description.

5.3 Data transfer

The last scenario we considered was devoted to the data transfer functionalities offered by the protocol, and in particular the "guaranteed mode" allowing data transmission even if the communication layer is not fully reliable. However, we first tried to verify it with a reliable communication layer, which is a necessary precondition.

The signals received by the protocol entities from the SSCF layer are the following:

- For the entity 1, the "AaEstablishRequest" signal and the data transfer requests of two distinct messages m_1 and m_2 ("AaDataRequest(m_1)" and "AaDataRequest(m_2)");
- For the entity 2, the "AaEstablishResponse" signal.

This signal set allows to build an asymmetrical scenario during which the connection can be established (upon entity 1 request), and transmission of message m_1 or m_2 can be requested at any time by entity 1. This asymmetry has been introduced in order to restrict the corresponding protocol behaviour, and the resulting state graph generated by OBJECTGÉODE contained 4 000 000 states, and 33 000 states after its reduction using ALDÉBARAN. The informal requirements we considered were the following:

Req 6: a data transfer *indication* is *never* transmitted by a protocol entity to its SSCF layer if it has not previously received a connection establishment *response* from this layer.

Req 7: a data transfer *indication* of a given message is *never* transmitted by a protocol entity to its SSCF layer if a data transfer *request* of the same message has not been previously received by the other entity.

Req 8: a data transfer *request* of a given message received by a protocol entity *is eventually* followed by a data transfer *indication* of the same message issued by the other entity.

Using EVALUATOR the evaluation on the protocol state graph of the μ-calculus version of these three requirements gave the following result:

- **Req 6** is true, which means that a connection is always correctly established when a data transfer occurs;

- **Req 7** is true, which means that there is no "message generation" performed by the protocol;
- **Req 8** happened to be false, and a diagnostic was produced by EVALUATOR.

The analysis of this diagnostic revealed something that seems to be an anomaly in the protocol behaviour described by the SDL specification. This anomaly concerns the "credit" value associated to a receiving entity, which records the number of messages that can be still received without sending back the corresponding acknowledgment. After acknowledgment this credit is then supposed to be reset to its initial value.

However, in the diagnostic sequence exhibited by EVALUATOR the credit value is never reset, which prevents the protocol to receive any further message once the initial credit has been reached. The connection is then released due to a timeout, and a new connection is established. This incorrect behaviour is clearly demonstrated when considering the "abstract" behaviour produced by ALDÉBARAN after minimization of the state graph with respect to branching bisimulation (where only "AaDataRequest" and "AaDataIndication" signal exchanges are observed).

5.4 Future work

The results obtained with this basic set of properties show that, even if they can be only partially applied by considering restrictive scenarios, model-checking verification techniques are quite useful to improve the knowledge of a system behaviour, or to detect some anomalies in its description.

Consequently this work needs to be continued, either by analyzing other scenarios (for instance the re-synchronization of a connection, the local data retrieval, etc.), or by considering a more unreliable environment for a protocol entity (including for instance an unreliable communication layer, possible failures of the other entity, etc.). However, it is likely the case in this last perspective that the state graph modeling the corresponding behaviour becomes too large to be fully generated. In these situations other facilities of the verification tools will have to be used, such as on-the-fly verification, or symbolic BDD-based representations [6].

6 Automatic generation of conformance tests

The SDL specification of the SSCOP protocol has been used for the automatic generation of test cases. This work has benefited from the preliminary analysis and optimizations made on the SDL specification. Verifications also gave us more confidence in the specification. This is important for automatic test generation as the specification is the reference model. Conversely, the first works made on test generation helped us in the process of specification correction and gave us some ideas on static analysis useful for verification and test generation.

Our objective in this case study was not to produce a "complete" test suite like those already available from the ATM Forum [8]. The first aim was to compare tests produced by TGV with those written by hand or produced by other tools. In particular we had the ambition to produce better tests from common test purposes, to treat more complex test purposes and to generate test cases for different test architectures. This case study was also the occasion to evaluate the maturity of our tool, to improve it and to open new research perspectives.

6.1 Tools used

The SDL toolset OBJECTGÉODE has been used for the edition (correction) of the SSCOP specification and for its simulation. ALDÉBARAN has been used with the explicit version of TGV for the τ^*-reduction, minimization and determinization of partial state graphs produced by OBJECTGÉODE. Some of these graphs and produced test cases have been visualized with a prototype tool named Viscope [17] which allows to draw state graphs in 2D or 3D. Finally, TGV has been used for test generation in its two use modes i.e. on explicit state graphs and on-the-fly with its connection to OBJECTGÉODE.

6.2 Preliminary analysis and test purpose formalization

A preliminary analysis of the specification (see section 4) allowed us to better understand the protocol and its SDL specification and to detect some transcription errors and possible simplifications.

The goal of this analysis was also to identify some interesting test purposes, to formalize them in order to generate test cases. Fifty test purposes have been identified and formally specified. These test purposes cover all functionalities of the SSCOP protocol but of course not all its possible behaviours. But most of these test purposes describe complex behaviours as they correspond to test cases covering several control states of the protocol (e.g. connection followed by disconnection, connection followed by data transfer, etc). This should be compared with the work made on verification of communicating SSCOP entities.

6.3 Analysis of available test suites

Several test suites have already been produced for the SSCOP protocol, such as the one produced by the tool TESTGEN (INT Evry France) [7]. But during this study we had only access to three TTCN test suites of the SSCOP protocol. These test suites had been produced in three different ways. We have tried to compare test cases produced by TGV with some test cases from those test suites. Test suites available to us were the following:

- the ATM Forum test suite (see ftp.atmforum.com, af-test-0067.000) has been written by hand by specialists of the SSCOP protocol. It is the richest test suite of the three considered ones because it reflects the expertise of test developers. It contains a declaration part (types of messages, timers definitions), a constraint part (values of message parameters, etc) a behaviour part which describes the sequencing of actions in each test case. These behaviours make full use of TTCN constructs such as loops, variables, separation of test cases into a preamble (a sequence leading to a particular control state), a test body (verifying the test purpose) an identification sequence (a sequence which can be used to identify the current control state of the protocol) and a postamble (return to the initial control state).

It is clear that some of the constructs used are difficult to generate automatically but we consider that this test suite represents a goal to reach by automatic tools.

- a test suite produced automatically by the SAMSTAG tool from the University of Lübeck [16] is also available. The generation is based on the description of test purposes by MSCs (Message Sequences Charts). The test suite also comprises a declaration part, a constraint part and a behaviour part. Behaviours are simpler that in the ATM Forum suite. In particular timers are not produced and one test case is basically a sequence leading to a PASS verdict, decorated with INCONCLUSIVE verdicts on undesired inputs. According to the paper, eight different versions of the SDL specification of the SSCOP have been used, each of them restricted to some functionalities of the protocol in order to be able to generate test cases.

- a test suite generated by TVÉDA from CNET. The available suite was produced by a previous version of TVÉDA called "syntactic TVÉDA". TVÉDA is limited to single process specifications. The tool automatically generates test purposes, by default one for each branch of each transition of the SDL specification. In this version of TVÉDA, preambles and postambles were not produced though they are with the new version. The computation of test cases was made by constraint resolution. The test suite contains a declaration part, a constraint part and a behaviour part.

  A new version of TVÉDA has also been used on SSCOP and produces more complete test cases but the test suite itself was not available to us but only a paper [9].

6.4 Test architecture

The test suites from TVÉDA and the ATM Forum consider that the tester has only access to the lower PCO. SAMSTAG considers that the two PCOs are observable and controllable. In fact, even for one PCO, most test cases need interactions through the upper PCO. This cannot be avoided as almost all control states of the protocol can only be accessed after some interactions through the upper PCO. Thus in TTCN test suites from TVÉDA and the ATM Forum, in the case of a non controllable PCO, these interactions are signaled with the mechanism of *implicit send.*

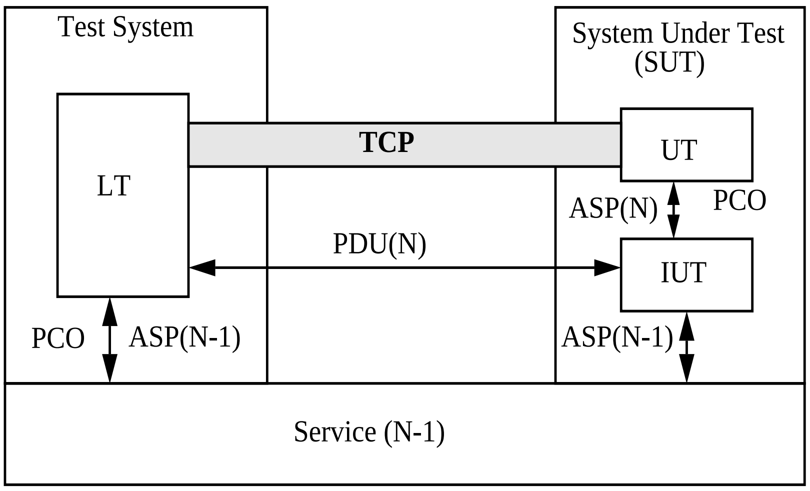

Figure 2: Remote testing architecture

The three above mentioned test suites are supposed to be derived for a Remote architecture (see figure 2). In fact this does not appear in test suites. In a remote test architecture one should see particular behaviours due to the asynchronism between the tester and the IUT. In fact the asynchronism is not taken into account. The test suite for a remote architecture seems to differ from a local test method only by the fact that PDUs (Protocol Data Units) and not ASP (Abstract Service Primitives) are exchanged with the lower tester.

Following these observations, we have decided to consider two different test architectures.

- a remote architecture with two PCOs and a synchronous interaction. This architecture is considered in order to compare produced test cases with the three available test suites with the same assumptions.
- a remote asynchronous architecture. Asynchronism is limited to the lower tester because we can suppose that the upper tester communicates in a synchronous way using ASPs: the synchronous abstraction is a good abstraction for this PCO. The lower tester communicates asynchronously, simulating a link in an ATM network. This communication is supposed not to be lossy as it is the tester itself that will simulate loss of data. In order to consider an asynchronous interaction between the protocol and its environment, we added a process between them. This adds a fifo queue between the specification of the SSCOP and the environment in each direction. The new process just delays interactions. Each message received from the environment (resp. from the SSCOP) is enqueued and later sent

to the SSCOP (resp. environment). This was necessary due to the communication semantics used in OBJECTGÉODE between the specification and the environment. This semantics states that messages received from or sent to the environment are not enqueued.

6.5 Experiments

TGV has been used in two ways, explicitly and on-the-fly. We detail here how these experiments were conducted and the results obtained.

6.5.1 Explicit TGV

When TGV is used in explicit mode, we first have to build the state graph of the specification with the OBJECTGÉODE simulator. But for a large specification as SSCOP (with a very large state graph), it is impossible to generate the complete state graph. Thus, for each test purpose, we have to build a partial state graph which allows to produce the corresponding test case. The first thing to do is to close the specification with inputs from the environment using the *feed* mechanism of OBJECTGÉODE, just as was done for verification. A subset of inputs is selected after a close look to the specification. These inputs are always available and are possible in several control states although they are ignored. We have thus used the mechanism of *stop conditions* in order to forbid these inputs in some states. Care must be taken to use *stop conditions* only in this context. In fact, *stop conditions* could be put on any transition, for example on outputs of the specifications, possibly producing biased test cases i.e. test cases that would reject correct implementations. A safer possibility is to use *refusal* states in the test purpose. But this was not available in TGV at the beginning of the study. After the state graph has been computed with OBJECTGÉODE, ALDÉBARAN minimizes it with respect to τ^*-a equivalence and determinizes it. This state graph represents the observable behaviour of the specification. TGV takes as inputs this state graph and the test purpose automaton and produces a test case which can be translated into TTCN.

This way of using TGV has been used only in the case of a synchronous communication between the IUT and the tester. At the time of this first experiment, the on-the-fly version of TGV was not available.

Fifty test cases have been produced corresponding to the fifty formalized test purposes. The sizes of the state graphs produced by OBJECTGÉODE were in the order of some thousands states. The reduction of these state graph by ALDÉBARAN produced state graphs of some hundred states. The total time spent for the generation of one test case was in the order of some seconds. Test cases produced by TGV for simple test purposes are quite comparable with those of the three available test suites. This allowed us to find some errors in those test suites such as bad management of timers or omission of inputs due to SSCOP timeouts.

6.5.2 On-the-fly generation

In the case of on-the-fly generation, TGV pilots OBJECTGÉODE and all phases (abstraction, τ^*-reduction and determinization) are done in one pass. This possibility of using TGV has been adopted for the two considered architectures.

Remote synchronous architecture: As mentioned earlier, the use of *stop conditions* has been suppressed and replaced by *refusal* states in test purposes. This allowed a simplification of test purposes descriptions and a better selection of test cases. On-the-fly generation also allows to relax constraints put by the environment and stop conditions in the case of explicit generation. Obtained test cases are generally identical to those produced in an explicit way. Differences may occur due to the exploration order and different constraints. But the global execution time is generally smaller as only a sub-graph of the specification is traversed and constructed by TGV.

Remote asynchronous architecture: As said previously, in this case the specification was completed with a new process which dissynchronizes the communication between the environment and the SSCOP protocol. In order to limit the behaviours of the new specification, we have limited to one the size of the queue associated to the channel from the environment to the specification. This can be justified by the fact that in practice, after sending a message to the IUT, the tester waits for reactions before sending a new message.

Produced test cases are often different from those produced in a synchronous communication context and are thus difficult to compare with available test suites. The main reason is that asynchronous interactions produces the classical problem of message collision. This happens very often as in many control states, after a first interaction and a timer setting, SSCOP waits for an input A and then sends B. But if A does not arrive in time, the timer expires and an output C is sent. Thus a tester sending A may receive either B or C. This is the case for example for a connection establishment (see the example below). Another typical situation may also happen due to asynchronism on multiple PCOs. The order in which messages are sent by the protocol is not necessarily conserved because messages can be delayed. Thus if the protocol entity sends A on a PCO followed by B on an other PCO, the tester should consider the possibilities of receiving A followed by B or B followed by A. The chosen testing architecture of SSCOP produces a derived situation as only the lower PCO is asynchronous. A situation which happens is then, when in a transition a message A is sent on the lower PCO followed by a message B on the upper PCO. In this case, we will always observe B before A.

Example: This last situation and a message collision happen in the following behaviour of SSCOP. In state *Idle*, when an *aaestablishrequest ASP* is received by SSCOP from the upper layer, a *bgninvoke PDU* is sent to the peer entity (the environment in our case), *timer_CC* is set and SSCOP goes to state *Outgoing Connection Pending.* In this state, SSCOP may receive several inputs among which a *bgaksignal PDU.* If this PDU is received, SSCOP sends an *aaestablishconfirm* to the upper layer. But if *timer_CC* expires, it may send again *bgninvoke.* After *Max_CC* timeouts of *timer_CC* and outputs of *bgninvoke* (in our example $Max\_CC = 4$) , SSCOP sends a message sequence composed of an *maaerrorindication* (which is considered unobservable here), an *endinvoke PDU* and a *aareleaseindication ASP* in this order.

In an asynchronous environment the behaviour of a tester which wants to envisage all the possible responses to a *bgaksignal* after an *aaestablishrequest* is quite complicated as proves the test produced by TGV in figure 3. The tester starts by sending an *aaestablishrequest*, receives a *bgninvoke PDU*, and sends a *bgaksignal.* Then it must wait for an *aaestablishconfirm* or a *bgninvoke PDU* due to message collision (timer_CC may have expired while *bgaksignal* is still progressing). The arrival of *bgaksignal* can be delayed for a long time, thus *timer_CC* may expire several times before it is received. The choice between receiving *aaestablishconfirm* or *bgninvoke PDU* is thus repeated twice (lines 4-5 and 6-7). After *Max_CC* - 1 receptions of *bgninvoke PDU* (line 7) it will have three possible continuations (lines 9, 13 and 14). First (line 13), it may receive an *aaestablishconfirm.* The second possibility (line 14) is to receive a last *bgninvoke PDU* followed either by a *release indication* (line 15) followed by an *endinvoke PDU* (due to the asynchronism on the lower PCO) or an *aaestablishconfirm* (line 18). But as the reception of *bgninvoke* may be delayed, a third possibility (line 8) is to receive an *aareleaseindication* before *bgninvoke* and *endinvoke.*

Despite a different testing architecture (only PDUs are controllable and observable), we can consider that the test case of the ATM Forum numbered S2_V_P3 partly corresponds to the previous example. It considers the output of *bgaksignal* by the tester in state Outgoing Connection Pending. The possibility to receive a *bgninvoke* is not considered, thus this event would lead to a fail verdict. This is either an error (the test case may reject a conformant implementation) or a proof that they suppose a synchronous communication in a remote testing architecture which is not realistic.

```
+---------------------------------------------------------------------------------------------------------------------+
|                                             Test Case Dynamic Behaviour                                             |
+---------------------------------------------------------------------------------------------------------------------+
| Test Case Name      : example                                                                                       |
| Group               :                                                                                               |
| Purpose             : Test the different response possibilities after a connection acknowledgement                 |
| Default             :                                                                                               |
| Comments            :                                                                                               |
+-------+-------+----------------------------------------------------+----------------------+------------+------------+
| Nr    | Label | Behaviour Description                              | Constraints Ref      | Verdict    | Comments   |
+-------+-------+----------------------------------------------------+----------------------+------------+------------+
|     1 |       | ut ! aaestablishrequest, St tbgninvoke             | aaestablishrequest0  |            |            |
|     2 |       |   lt ? bgninvoke, Cl tbgninvoke                    | bgninvoke1           |            |            |
|     3 |       |     lt ! bgaksignal,                               | bgaksignal2          |            |            |
|       |       |          St tbgninvoke, St taaestablishconfirm     |                      |            |            |
|     4 |       |       ut ? aaestablishconfirm,                     |                      |            |            |
|       |       |            Cl taaestablishconfirm, Cl tbgninvoke   |                      | (PASS)     |            |
|     5 |       |       lt ? bgninvoke,                              | bgninvoke1           |            |            |
|       |       |            Cl taaestablishconfirm, Cl tbgninvoke,  |                      |            |            |
|       |       |            St tbgninvoke, St taaestablishconfirm   |                      |            |            |
|     6 |       |         ut ? aaestablishconfirm,                   |                      |            |            |
|       |       |              Cl taaestablishconfirm, Cl tbgninvoke |                      | (PASS)     |            |
|     7 |       |         lt ? bgninvoke,                            | bgninvoke1           |            |            |
|       |       |              Cl taaestablishconfirm, Cl tbgninvoke,|                      |            |            |
|       |       |              St tbgninvoke, St taaestablishconfirm,|                      |            |            |
|       |       |              St taareleaseindication               |                      |            |            |
|     8 |       |           ut ? aareleaseindication,                | aareleaseindication3 |            |            |
|       |       |                Cl taareleaseindication,            |                      |            |            |
|       |       |                Cl taaestablishconfirm,             |                      |            |            |
|       |       |                Cl tbgninvoke,                      |                      |            |            |
|       |       |                St tbgninvoke                       |                      |            |            |
|     9 |       |             lt ? bgninvoke,                        | bgninvoke1           |            |            |
|       |       |                  Cl tbgninvoke, St tendinvoke      |                      |            |            |
|    10 |       |               lt ? endinvoke, Cl tendinvoke        | endinvoke4           | (PASS)     |            |
|    11 |       |               ? tendinvoke                         |                      | FAIL       |            |
|    12 |       |             ? tbgninvoke                           |                      | FAIL       |            |
|    13 |       |           ut ? aaestablishconfirm,                 |                      |            |            |
|       |       |                Cl taareleaseindication,            |                      |            |            |
|       |       |                Cl taaestablishconfirm,             |                      |            |            |
|       |       |                Cl tbgninvoke                       |                      | (PASS)     |            |
|    14 |       |           lt ? bgninvoke,                          | bgninvoke1           |            |            |
|       |       |                Cl taareleaseindication,            |                      |            |            |
|       |       |                Cl taaestablishconfirm,             |                      |            |            |
|       |       |                Cl tbgninvoke,                      |                      |            |            |
|       |       |                St taaestablishconfirm,             |                      |            |            |
|       |       |                St taareleaseindication             |                      |            |            |
|    15 |       |             ut ? aareleaseindication,              | aareleaseindication3 |            |            |
|       |       |                  Cl taareleaseindication,          |                      |            |            |
|       |       |                  Cl taaestablishconfirm,           |                      |            |            |
|       |       |                  St tendinvoke                     |                      |            |            |
|    16 |       |               lt ? endinvoke, Cl tendinvoke        | endinvoke4           | (PASS)     |            |
|    17 |       |               ? tendinvoke                         |                      | FAIL       |            |
|    18 |       |             ut ? aaestablishconfirm,               |                      |            |            |
|       |       |                  Cl taareleaseindication,          |                      |            |            |
|       |       |                  Cl taaestablishconfirm            |                      | (PASS)     |            |
|    19 |       |             ? taareleaseindication                 |                      | FAIL       |            |
|    20 |       |             ? taaestablishconfirm                  |                      | FAIL       |            |
|    21 |       |           ? taareleaseindication                   |                      | FAIL       |            |
|    22 |       |           ? taaestablishconfirm                    |                      | FAIL       |            |
|    23 |       |           ? tbgninvoke                             |                      | FAIL       |            |
|    24 |       |         ? taaestablishconfirm                      |                      | FAIL       |            |
|    25 |       |         ? tbgninvoke                               |                      | FAIL       |            |
|    26 |       |       ? taaestablishconfirm                        |                      | FAIL       |            |
|    27 |       |       ? tbgninvoke                                 |                      | FAIL       |            |
|    28 |       |   ? tbgninvoke                                     |                      | FAIL       |            |
+-------+-------+----------------------------------------------------+----------------------+------------+------------+
```

Figure 3: A test case generated by TGV for a remote asynchronous architecture

This example makes evident the need of using automatic tools as human mind has some difficulties to envisage all possible behaviours in complex situations such as the one presented above, and this may cause many errors in manual test cases. The advantage of TGV on other tools is crucial for this kind of situations. First, contrary to some other tools (TVÉDA for example), TGV is not limited to one process. Thus modeling different testing architectures by extension of the specification is compatible with test generation. Second, TGV produces test cases which can have several branches leading to a PASS verdict.

To our knowledge, TGV is the only tool that can make this. All other tools are based on the computation of one main sequence of the observable behaviour of the specification. In the case were different outputs are possible, the tester has to consider all possible inputs. In these tools, one possibility is continued and lead to a PASS verdict while all other possible inputs immediately produce an INCONCLUSIVE verdict. This is too restrictive, especially in the case of asynchronism were several possible arrival orders should be considered equally. This is very important for test execution too because test cases should be reexecuted until a PASS or FAIL verdict is reached, Thus INCONCLUSIVE verdicts should be avoided, when continuations may lead to a PASS verdict. This principle is adopted by TGV.

6.5.3 Verification combined with test generation

At the beginning of our experiments, as we still had doubts on the SDL specification used, we have used verification capabilities of OBJECTGÉODE while generating tests with TGV. We have encoded in a GOAL observer an automaton describing the abstract behaviour of the SSCOP protocol at its upper interface SSCF. The test generation is made on a synchronous product of the specification and the observer. Thus we verify that all sequences traveled during the test generation are at least accepted by this automaton. This gives more confidence in the specification and in the generated test cases.

Another observer, an intrusive one, was also used to reduce the size of the state graph. The role of this observer was to reset the implicit variable SENDER which is never used in the specification (see section 4).

6.5.4 Future work in test generation

As TGV is still a prototype, the work made on case studies as SSCOP helps us to improve it. In particular, we are designing a new generation algorithm which will produce test cases with loops and, as a consequence, still less INCONCLUSIVE verdicts. This is particularly interesting in current situations were some inputs may happen without modifying the expected behaviour. We are also working on the expressive power of test purposes in order to allow more abstraction on parameters and the possibility to describe more discriminating test purposes with unobservable actions and states predicates. These improvements will be implemented in TGV and tested on the SSCOP specification.

The SSCOP specification has a large control part but also a large data part. TGV treats data by enumeration and this obliges us to limit the variables domains or fix the parameters of interactions. This encourages us to have a closer look at symbolic methods and proof methods. Symbolic methods could avoid enumeration and used in conjunction with proof methods and classical verification methods, we expect to produce test cases closer to manual ones i.e. which also manipulate variables (counters for example) which are common in TTCN.

We are also investigating the problem of distributed testing. The literature on the subject is rather poor because it is a difficult subject. But Concurrent

TTCN the new version of TTCN allows to describe distributed testers and test suites for multi-party testing already exist. Thus, users of test generation tools will soon want to generate distributed testers. We are particularly interested by this research and we have made first steps in the direction of producing distributed tests.

Finally, we are working on an industrial project with Verilog and Cnet which aim is to develop an industrial test generation tool in the ObjectGéode environment. This tool will be adapted from three tools: TVéda from Cnet, TTCgeN from Verilog and Tgv.

7 Conclusion and future work

This case study is a representative one of a large class of protocols. The complexity of the data part leads to combine other approaches with model-checking. The use of data-flow (or static) analysis, originally a component of global optimization part of a compiler, in the context of model-checking, allows to abstract the data part with respect to the desired property.

The work done on the Sscop protocol has been very interesting on many aspects. It was rapidly clear that brute force verification could not work on the original specification due to its inherent complexity. This statement led us to the study of techniques for the reduction of this complexity, before the application of brute force tools.

- Static analysis proved very useful for the reduction of state graphs which is profitable for the purpose of verification as well as for test generation.
- Verification on abstract state graphs obtained without variable evaluation allowed to detect subtle errors in the Sdl specification.
- Defining restricted environments instead of completely chaotic environments allowed to prove basic properties and to detect an error in a system composed of a pair of communicating entities.
- The on-the-fly technique, especially for test generation, proved again its efficiency even on such a large specification.

Another lesson of this case study is the strong link between verification and test generation. Confidence in the specification is crucial for test generation as it is used as the reference model. Thus our work on verification, even if it is partial, has been very useful for the confidence in generated test cases. Moreover, as said above, both activities take benefit of all optimizations made by static analysis on the specification. An interesting aspect is also the use of ObjectGéode observers during test generation. This allowed to perform optimizations and to verify that produced test cases are correct with respect to an abstract behaviour of the Sscop protocol. This again improves the confidence in generated test cases. A last point to notice is the great similarity between some properties that have been verified on peer entities and some test purposes used for test generation. As algorithms are quite similar, this is another proof of the great interaction between these activities which deserves further developments.

Last, but not least, it allowed us to improve our tools and to develop new ones. In particular several tools have been slightly improved by their connection to ObjectGéode and consequently their ability to treat Sdl specifications. Evaluator and Tgv are now connected to the simulator API. This allows Evaluator to perform on-the-fly model checking and Tgv to generate on-the-fly test cases from Sdl specifications. The development of a new static analysis tool connected to the API of ObjectGéode's compiler through Sdl2aut now allows to perform static analysis on Sdl specifications.

The case study provider (Cnet) expressed a great interest for the results obtained on the Sscop specification and test cases. The specification has been slightly improved by numerous optimizations and corrections of detected errors. The two first test generation campaigns with a synchronous interaction and their comparison with available test suites has allowed to detect some errors in the different test suites. The experiment with asynchronous interactions has produced interesting test cases. These results proved again that automation is profitable in quality for complex specifications.

Our work on a complex case study such as Sscop has been very fruitful also for the numerous research perspectives open or confirmed. The first experiments on static analysis for the optimization of specifications have been very encouraging and deserves further developments. The idea of using supplementary information such as the property to check or the test purpose seems promising for a more efficient analysis in the perspective of model-checking or test generation. The improvement of our model-checking and test generation algorithms is also a constant concern and the present case study has given us some new ideas on such improvements. This is particularly important in order to produce test cases of better quality. For this aim, we are also starting to work on the conjunction of different methods such as symbolic methods, proof methods and test generation, with the ambition to generate parametrized test cases which manipulate data. In parallel we have already started to work on the difficult problem of distributed testing which needs knowledge in testing, distributed systems and program transformation. And finally, as already noticed, this case study showed us that the interaction between verification and test generation needs further work which will certainly be fruitful for both activities.

References

[1] ISO/IEC International Standard 8807. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical report, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[2] ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - Part 2 : Abstract Test Suite Specification - Part 3 : The Tree and Tabular Combined Notation (TTCN), 1992.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[4] B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *SDL forum'95.* Elsevier Science (North Holland), 1995.

[5] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for Branching Time Semantics. In *18th ICALP.* Springer Verlag, july 1991.

[6] M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the ALDÉBARAN Toolset. *First edition of the STTT (Software Tools and Technology Transfer) journal*, 1997.

[7] A. Cavalli, B.-H. Lee, and T. Macavei. Test generation for the SSCOP-ATM networks protocol. In *Proceedings of SDL forum'97.* Elsevier Science (North Holland), 1997.

[8] The ATM Forum Technical Committee. Conformance abstract test suite for the SSCOP for UNI 3.1, af-test-0067.000, sept 1996. Available by ftp at ftp.atmforum.com.

[9] I. Disenmayer, S. Gauthier, and L. Boullier. L'outil TVEDA dans une chaîne de production de tests d'un protocole de télécommunication. In G. Leduc, editor, *CFIP'97 : Ingénierie des Protocoles*, pages 271–286. Hermès, sept 1997.

[10] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification, CAV'96 (New Brunswick, New Jersey, USA).* LNCS 1102 Springer Verlag, August 1996.

[11] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification, CAV'96, (New Brunswick, New Jersey, USA).* LNCS 1102 Springer Verlag, aug 1996.

[12] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocoles with Verification Technology. *Science of Computer Programming*, 29, 1997.

[13] J.-C. Fernandez and L. Mounier. A Local Checking Algorithm for Boolean Equation Systems. Technical Report Spectre-95-07, Verimag, Grenoble-France, 1995.

[14] H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation and Testing. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer-Verlag, April 1998.

[15] R.J. Van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (extended abstract). CS-R 8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989.

[16] J. Grabowski, R. Scheurer, and D. Hogrefe. Applying SAMSTAG to the B-ISDN Protocol SSCOP. Technical Report A-97-01, part I, University of Lübeck, January 97.

[17] T. Jéron and C. Jard. 3D layout of reachability graphs of communicating processes. In *Graph Drawing'94, DIMACS Workshop*, pages 25–33, Princeton, New-Jersey, Octobre 1994. LNCS n^o 894. Paru en rapport de recherche bilingue français-anglais, Irisa n^o 852 et Inria n^o 2334.

[18] T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée: application à la génération de test. In G. Leduc, editor, *CFIP'97 : Ingénierie des Protocoles*. Hermes, sept 1997.

[19] A. Kerbrat, C. Rodriguez, and Y. Lejeune. Interconnecting the OBJECT-GÉODE and CÆSAR-ALDÉBARAN Toolsets. In *Proceedings of SDL forum'97*. Elsevier Science (North Holland), 1997.

[20] D. Kozen. Results on the Propositional μ-Calculus. In *Theoretical Computer Science*. North-Holland, 1983.

[21] R. Milner. A Calculus of Communication Systems. In *LNCS 92*. Springer Verlag, 1980.

[22] ITU-T Recommendation Q.2110. B-ISDN - ATM Adaptation Layer - Service Specific Connection Oriented Protocol (SSCOP), 1994.

[23] ITU-T Recommendation Q.2130. Couche d'adaptation du mode de transfert asynchrone de signalisation dans le RNIS à large bande - fonction de coordination propre au service pour la signalisation à l'interface utilisateur-réseau, 1994.

[24] ITU-T Recommendation Z-100. Specification and Description Language, 1996.

[25] ITU-T Recommendation Z-120. Message Sequence Charts, 1996.

[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Edyy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Inc., Englewood Cliffs, 1991.

[27] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[28] VERILOG. ObjectGeode SDL Simulator Reference Manual. Technical report, VERILOG, 1996.

[29] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.