Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

TGV: theory, principles and algorithms

Claude Jard **Thierry Jéron**
IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
jard@irisa.fr **jeron@irisa.fr**

ABSTRACT: This paper presents the TGV tool allowing the automatic synthesis of conformance test cases from a formal specification of a reactive system. TGV has been developped by Irisa Rennes and Verimag Grenoble, with the support of the Vasy team of Inria Rhônes-Alpes. The paper describes the main elements of the underlying testing theory, based on a model of transitions system which distinguishes inputs, outputs and internal actions, and based on the concept of conformance relation. The principles of the test synthesis process are explained as well as the main algorithms. We then describe the main characteristics of the TGV tool. As a conclusion, we describe some on going works in test synthesis.

I. Conformance testing

Testing, in all its variations, is one of the most used validation techniques. In this paper, we focus on *conformance testing* applied to reactive systems. By reactive system, we mean a software component which reacts to stimuli of its environment. Conformance testing consists in checking that the behavior of a real implementation of a reactive system (IUT for *Implementation Under Test*) is correct with respect to a specification. The code of the IUT is unknown, and its behavior is only visible by interaction with a tester which controls and observes the IUT through dedicated interfaces (called PCO for *Points of Control and Observation*). Conformance testing is a type of functional testing of a black box nature.

A. Some basic concepts

In the context of telecommunication protocols, the main concepts of this activity are described in the standard ISO 9646 [1]. Some of them are introduced here.

A *test case* is an elementary test targeted to testing a particular functionality, called *test purpose*. A *test suite* is a set of test cases. The basic elements of a test case are interactions through PCO: outputs are stimuli sent in order to control the IUT's input events, inputs are observations of the IUT's output events. Inputs may lead to different *verdicts*. A Fail verdict denotes a divergence with the expected behaviour and the IUT is rejected. A Pass verdict is returned if the observation is correct and the test purpose is reached. An Inconclusive verdict is returned if a correct behavior is observed, but it is impossible to reach the test purpose. This is due to the fact that, in general, reactive systems cannot be completely controlled by a tester: it may have several outputs to the same input. The *tester*, specialized hardware, software or human operator, executes test cases. But as test cases are often described with some abstraction level (they are called *abstract test cases*), they must be translated into *executable test cases*.

B. Formalizing for automation

Conformance testing is a costly activity which takes an important part in the global cost of a software. For a long time, the scientific community tries to automate the process of deriving test cases. For conformance testing, the reference behaviour is described by the specification which determines the verdicts. Automation thus induces formalizing the specification, but also formalizing the interaction between the tester and the IUT. The definition of verdicts also forces to formalize conformance i.e. the relation between the IUT and its specification that is checked during testing. Algorithms for the automatic test case synthesis, which take specifications as inputs, must be designed. Essential properties of produced test cases must be established. *Soundness* means that test cases only reject non conformant IUT, *exhaustiveness* means that all non conformant implementations are rejected by a test suite. The main ingredients for automation are described in [27].

The paper is organized as follows. In section II we present the underlying testing theory of TGV. Then section III presents the synthesis algorithms. The TGV tool is described in section IV. Finally we conclude and draw some perspectives in section V.

II. Testing theory in TGV

The contribution of TGV in automatic synthesis of test cases is mainly in the area of algorithms and tool. However TGV is based on a conformance testing theory, inspired from works around Jan Tretmans (University of Twente) [26]. This theory inherits from preceding works on testing equivalence and preoders [6], [2]. The behaviours of specifications and IUT are modelled by a variant of labelled transition systems (LTS). Roughly speaking, the conformance relation is a partial inclusion of traces of observable events and quiescence. We now present this theory, adapted to make it more effective and understandable.

A. Modelling with transition systems

For conformance testing, a distinction must be made between events of the system that are *controllable* by the environment (the inputs), from those that are only *observable* (the outputs). The model we adopt (called IOLTS for Input-output LTS) is an adaptation of the classical LTS model.

Definition 1: An IOLTS is a quadruple $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$ where Q^{M} is a finite non-empty set of states, $q_0^{\mathrm{M}} \in Q^{\mathrm{M}}$ is the initial state, A^{M} is the alphabet of actions. It is partitioned into three sets $A^{\mathrm{M}} = A_{\mathrm{I}}^{\mathrm{M}} \cup A_{\mathrm{O}}^{\mathrm{M}} \cup I^{\mathrm{M}}$. $A_{\mathrm{I}}^{\mathrm{M}}$ is the input alphabet, $A_{\mathrm{O}}^{\mathrm{M}}$ is the output alphabet, and I^{M} the alphabet of internal actions. $\rightarrow_{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A^{\mathrm{M}} \times Q^{\mathrm{M}}$ is the transition relation.

Note: For the sake of clarity, in the examples, we will note $?a$ for an input $a \in A_{\mathrm{I}}^{\mathrm{M}}$ and $!x$ for an output $x \in A_{\mathrm{O}}^{\mathrm{M}}$.

Notations: Let $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$ be an IOLTS. The subscript (or superscript) $_{\mathrm{M}}$ will be omitted when clear from the context. We write $q \xrightarrow{a}_{\mathrm{M}} q'$ for $(q, a, q') \in \rightarrow_{\mathrm{M}}$ and $q \xrightarrow{a}_{\mathrm{M}}$ for $\exists q' : q \xrightarrow{a}_{\mathrm{M}} q'$. An $IOLTS$ is sometimes denoted by its initial state and we write $M \rightarrow_{\mathrm{M}}$ for $q_0^{\mathrm{M}} \rightarrow_{\mathrm{M}}$. Let $\mu_{(i)} \in A^{\mathrm{M}}$ some actions, $a_{(i)} \in A^{\mathrm{M}} \setminus I^{\mathrm{M}}$ some visible actions (inputs or outputs), $\tau_{(i)} \in I^{\mathrm{M}}$ some internal actions $\sigma \in (A^{\mathrm{M}} \setminus I^{\mathrm{M}})^*$ a sequence of visible actions, $q, q' \in Q^{\mathrm{M}}$ some states.
$\Gamma(q) \triangleq \{\mu \in A^{\mathrm{M}} \mid q \xrightarrow{\mu}_{\mathrm{M}}\}$ is the set of fireable actions in q. $Out_{\mathrm{M}}(q) \triangleq \Gamma(q) \cap A_{\mathrm{O}}^{\mathrm{M}}$ is the set of fireable outputs in q. We extend it to sets of states: for $P \subseteq Q^{\mathrm{M}}$
$Out_{\mathrm{M}}(P) \triangleq \{Out_{\mathrm{M}}(q) \mid q \in P\}$.

We note $q \xrightarrow{\mu_1 \cdots \mu_n}_{\mathrm{M}} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} q_n = q'$.

Visible behaviours are described by the $\Rightarrow$ relation. We note $q \overset{\varepsilon}{\Rightarrow} q' \triangleq q = q'$ or $q \xrightarrow{\tau_1 . \tau_2 \cdots \tau_n}{}^{*} q'$ and $q \overset{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 : q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\varepsilon}{\Rightarrow} q'$. We also use the notations $q \overset{a_1 \cdots a_n}{\Rightarrow} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{a_n}{\Rightarrow} q_n = q'$ and $q \overset{\sigma}{\Rightarrow} \triangleq \exists q' : q \overset{\sigma}{\Rightarrow} q'$. The set q after $\sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$ (resp. P after $\sigma \triangleq \bigcup_{q \in P} q$ after σ) is the set of states reachable from q (resp. from the state set P) by action sequences from which only the projection σ onto visible actions is defined. $Traces(q) \triangleq \{\sigma \in (A \setminus I)^* \mid q \overset{\sigma}{\Rightarrow}\}$ (resp. $Traces(M) \triangleq Traces(q_0^{\mathrm{M}})$) describe the sequences of visible action fireable from q (resp. from the initial state of an IOLTS M).

The relation $\Rightarrow$ defines an IOLTS with same traces as M but deterministic (with no visible actions).

Definition 2: The *deterministic IOLTS of* $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$, denoted by $det(M)$ is a deterministic IOLTS defined by
$det(M) = (2^{Q^{\mathrm{M}}}, A^{\mathrm{M}} \setminus I^{\mathrm{M}}, \Rightarrow, q_0^{\mathrm{M}} \text{ after } \varepsilon)$.

States of $det(M)$, called *meta-states* in the sequel, are subsets of Q^{M}, the initial state q_0^{M} after ε is the set of states reachable from q_0^{M} by internal actions. In section III-C we will see an efficient construction of this IOLTS.

Models of specifications: A specification of a reactive system is in general given in a specialized language or notation (SDL, Lotos, UML, IF in the case of TGV). The operational semantics of such a language, implemented in a simulator, describes all possible behaviors of specifications. We suppose here that the semantics of a specification is given by an IOLTS $S = (Q^{\mathrm{S}}, A^{\mathrm{S}}, \rightarrow_{\mathrm{S}}, q_0^{\mathrm{S}})$. The example given in the left side of figure 1 will be our running example.

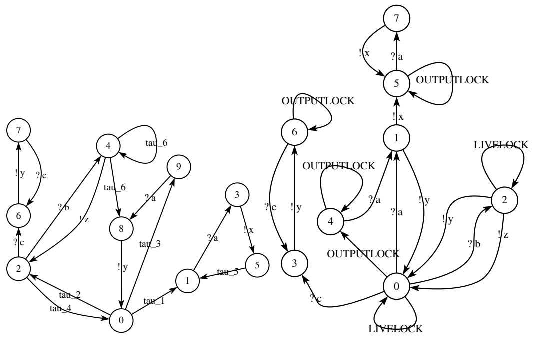

Fig. 1. Specification S and $det(\delta(S))$, its visible behaviour

Models of implementations: The implementation under test (IUT) is a black box interacting with a tester. It is not a formal object. However, if we want to reason about conformance, we have to consider that the IUT's behaviours can be modelled. This is called the *test hypothesis*.

An IUT is modelled by an IOLTS $IUT = (Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, \rightarrow_{\mathrm{IUT}}, q_0^{\mathrm{IUT}})$ with $A^{\mathrm{IUT}} = A_{\mathrm{I}}^{\mathrm{IUT}} \cup A_{\mathrm{O}}^{\mathrm{IUT}} \cup I^{\mathrm{IUT}}$. We will always suppose the compatibility of the alphabets of the IUT and S i.e. $A_{\mathrm{I}}^{\mathrm{S}} \subseteq A_{\mathrm{I}}^{\mathrm{IUT}}$, and $A_{\mathrm{O}}^{\mathrm{S}} \subseteq A_{\mathrm{O}}^{\mathrm{IUT}}$. We assume that the IUT is *input complete*: in each state all inputs are accepted i.e. $\forall q \in Q^{\mathrm{IUT}}, \forall a \in A_{\mathrm{I}}^{\mathrm{IUT}}, q \xrightarrow{a}$. This hypothesis is reasonable when the IUT never refuses an invalid or inopportune input but answers negatively.

Quiescence: In practice, tests observe traces of a system but also quiescence by *timers*. Several kinds of quiescence may happen and are illustrated in the left side of figure 2: *deadlock*: the system cannot evolve i.e. $\Gamma(q) = \emptyset$, *output quiescence*: the system is waiting for an input from the environment i.e. $\Gamma(q) \subseteq A_{\mathrm{I}}^{\mathrm{M}}$, or *livelock*: the system diverges by an infinite sequence of internal actions. In the case of finite state systems that we consider, a livelock is a loop of internal actions i.e. $\exists \tau_1, \tau_2, \ldots \tau_n, q \overset{\tau_1.\tau_2 \cdots \tau_n}{\rightarrow} q$.

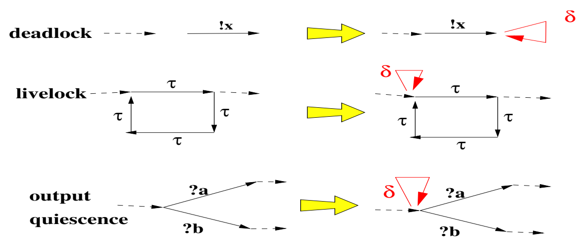

Fig. 2. Quiescence and how to explicit it

As conformance testing is based on the observation of visible behaviours, test synthesis requires a determinization of the specification: two sequences with same traces cannot be distinguished, but their respective suffix must be considered as possible evolutions of the system. Also, the information about quiescence of the specification must be preserved by determinization. This is only possible if quiescence is computed on the specification as described in the sequel and sketched in figure 2.

Definition 3: The *suspension automaton* of an IOLTS $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \rightarrow_{\mathrm{M}}, q_0^{\mathrm{M}})$ is an IOLTS $\delta(M) = (Q^{\mathrm{M}}, A^{\delta(\mathrm{M})}, \rightarrow_{\delta(\mathrm{M})}, q_0^{\mathrm{M}})$ where $A^{\delta(\mathrm{M})} = A^{\mathrm{M}} \cup \{\delta\}$ with $\delta \in A_{\mathrm{O}}^{\delta(\mathrm{M})}$ (δ is considered as an output, observable by the environment), and the transition relation $\rightarrow_{\delta(\mathrm{M})}$ is obtained from $\rightarrow_{\mathrm{M}}$ by adding loops $q \overset{\delta}{\rightarrow} q$ for each quiescent state q (i.e. deadlock, livelock or output quiescence).
The right side of figure 1 presents $det(\delta(S))$, the visible behavior of the specification example S. Note that δ actions are distinguished.

B. Conformance relation

A *conformance relation* formalizes the set of IUT that are considered as correct w.r.t a specification. Following Tretmans [26], the considered observations are traces of the suspension automaton (traces with quiescences). Intuitively, an implementation IUT conforms to its specification S for **ioco** if after each trace σ of $\delta(S)$ the IUT only exhibits outputs and quiescences possible in S. Formally:

Definition 4: Let S be an IOLTS and IUT be an input complete IOLTS (compatible with S),
IUT **ioco** $S \triangleq \forall \sigma \in Traces(\delta(S))$,
$Out(\delta(IUT)$ after $\sigma) \subseteq Out(\delta(S)$ after $\sigma)$

Examples : Figure 3 explains **ioco**. IUT_1 **ioco** S because in each state, outputs of IUT_1 are included in outputs of S. **ioco** thus allows to restrict the IUT on outputs. IUT_1 **ioco** S also even if the initial state of IUT_2 allows a new input $?b$ as only the outputs are checked by **ioco**. **ioco** thus allows partial specifications. However $\neg(IUT_2$ **ioco** $S)$ as the output $!z$ after the input $?a$ is not allowed in the specification. and the quiescence after $?a.!x$ (due to an internal loop for example) is not specified in S.

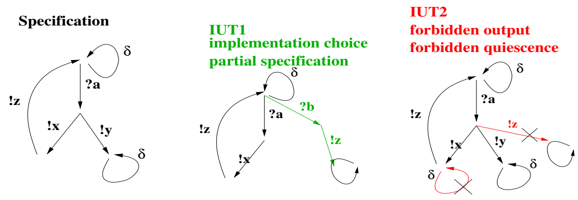

Fig. 3. **ioco** by the example

C. Tests: models, execution and properties

Reactive systems that we consider are not always controllable by their environment. Thus test cases should have the choice between correct inputs and should foresee non-conformant IUT. However, they have no choice between outputs as they control them. And they have no internal actions. To model test cases, we choose IOLTS with verdicts and some additional properties. A test case has a complex behavior which structure is a graph, with possible loops.

Definition 5: A *test case* is an IOLTS $TC = (Q^{\mathrm{TC}}, A^{\mathrm{TC}}, \rightarrow_{\mathrm{TC}}, q_0^{\mathrm{TC}})$ equipped with three sets of trap states **Pass** $\subseteq Q^{\mathrm{TC}}$, **Fail** $\subseteq Q^{\mathrm{TC}}$ and **Inconc** $\subseteq Q^{\mathrm{TC}}$ characterizing verdicts. Its alphabet is $A^{\mathrm{TC}} = A_{\mathrm{I}}^{\mathrm{TC}} \cup A_{\mathrm{O}}^{\mathrm{TC}}$ where $A_{\mathrm{O}}^{\mathrm{TC}} \subseteq A_{\mathrm{I}}^{\mathrm{S}}$ (TC emits only inputs of S) and $A_{\mathrm{I}}^{\mathrm{TC}} \subseteq A_{\mathrm{O}}^{\mathrm{IUT}} \cup \{\delta\}$ (TC foresee any output or quiescence of IUT). By hypothesis, states in **Fail** and **Inconc** are directly reachable only by inputs: $\forall (q, a, q') \in \rightarrow_{\mathrm{TC}}$, ($q' \in$ **Inconc** $\cup$ **Fail** $\Rightarrow a \in A_{\mathrm{I}}^{\mathrm{TC}}$), and from each state, a verdict is reachable $\forall q, \exists \sigma \in A^{\mathrm{TC}*}, \exists q' \in$ **Pass**$\cup$ **Inconc** $\cup$ **Fail**, $q \overset{\sigma}{\rightarrow} q'$. TC is *controllable*: no choice is allowed between two outputs or an input and output, i.e. $\forall q \in Q^{\mathrm{TC}}, \forall a \in A_{\mathrm{O}}^{\mathrm{TC}}, q \overset{a}{\rightarrow}_{\mathrm{TC}} \Rightarrow \forall b \neq a, q \overset{b}{\not\rightarrow}_{\mathrm{TC}}$. It is *input complete* in all states where an input is possible: $\forall q \in Q^{\mathrm{TC}}$, $(\exists\, a \in A_{\mathrm{I}}^{\mathrm{TC}},\ q \overset{a}{\rightarrow}_{\mathrm{TC}} \Rightarrow \forall\, b \in A_{\mathrm{I}}^{\mathrm{TC}},\ q \overset{a}{\rightarrow}_{\mathrm{TC}})$.
A *test suite* is a set of test cases.

Test cases are executed on an IUT. This execution is modelled by a parallel composition with a synchronization on common visible actions:

$$\frac{p \overset{a}{\rightarrow}_P p',\, q \overset{a}{\rightarrow}_Q q'}{(p,q) \overset{a}{\rightarrow}_{P||Q} (p',q')} \qquad \frac{p \overset{\tau}{\rightarrow}_P p'}{(p,q) \overset{\tau}{\rightarrow}_{P||Q} (p',q)} \qquad \frac{q \overset{\tau}{\rightarrow}_Q q'}{(p,q) \overset{\tau}{\rightarrow}_{P||Q} (p,q')}$$

This model of execution, together with the hypothesis made on the IUT and test cases, ensures that $TC||IUT$ may only block in states where a verdict is returned by TC. Thus verdicts are associated to *maximal traces* of the test cases i.e. sequences $\sigma \in A^{\mathrm{TC}^*}$ such that $\Gamma(q_0^{\mathrm{TC}}$ after $\sigma) = \emptyset$. Note that test cases (in particular those generated by TGV) may have loops.

Thus test execution may be infinite. To prevent this, global timers should be used.

A *verdict* associated to the execution of a test case TC on an IUT is completely determined by the state of TC reached by a maximal trace of $IUT||TC$. Depending on this state, it can be *Pass*, *Fail* or *Inconc*. $verdict(\sigma) = Fail$ $(resp.\ Pass, Inconc) \triangleq$ TC after $\sigma \subseteq$ **Fail** $(resp.$ **Pass**, **Inconc**$)$

A possible rejection of an IUT by a test case is defined by: $TC\ may\ reject\ IUT \triangleq \exists \sigma \in Traces(TC||IUT),\ verdict(\sigma) = Fail$.
may pass and *may inconc* are defined in the same way. Notice that the lack of control of test cases on an IUT implies that a unique test case may reject, accept or return an inconclusive verdict on the same IUT.

Definition 6: A test case TC is *sound* for S and **ioco** if $\forall IUT, IUT$ **ioco** $S \Rightarrow \neg(TC\ may\ reject\ IUT)$.
A test suite is sound if it consists of sound test cases.

A test suite is *exhaustive* for S and **ioco** if
$\forall IUT, \neg(IUT$ **ioco** $S) \Rightarrow TC\ may\ reject\ IUT$
A test suite is *complete* if it is sound and exhaustive.

The minimal property required for test suites is *soundness*: a test suite should not reject a conformant IUT. This property is reachable, but not sufficient in practice as test cases accepting all IUT are sound. One would like *exhaustive* test suites i.e. every non conformant IUT would be rejected. But it is unreachable for finite test suites as soon as the specification has loops. It requires an infinite number of test cases or infinite state test cases. Thus we will only require the exhaustiveness of the synthesis technique: the infinite test suite composed of all test cases that the synthesis algorithm can construct is exhaustive. Thus, for non conformant IUT, it is theoretically possible to produce a test case that may reject it.

D. Formal test purposes

One of the main ingredients of the test synthesis technique implemented in TGV, is the formalization of the concept of test purpose, and its use for test selection. In practice, test purposes are informal descriptions of behaviors to be tested, in general incomplete sequences of actions. We model test pruposes by automata (IOLTS extended with marked states) accepting sequences of actions of the specification. To allow an efficient selection, in particular on-the-fly (see section III-F), two distinct sets of marked states are considered, which accept or refuse sequences of actions of the specification.

Definition 7: A *test purpose* is a deterministic and *complete* IOLTS $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$, equipped with two sets of *trap states* $Accept^{TP}$ and $Refuse^{TP}$, with same alphabet as the specification i.e. $A^{TP} = A^{S}$. *Complete* means that each state allows all actions i.e. $\forall q \in Q^{TP}, \forall \in A^{TP}, q \xrightarrow{a}_{TP}$ and a *trap* state q has a loops on each action i.e. $\forall a \in A^{TP}, q \xrightarrow{a}_{TP} q$.

Note and example: It is interesting to allow abstraction in the description of test purposes, with respect to the specification behaviour. To satisfy the completeness requirement, we use the label "*" in TGV which, in a transition $q \xrightarrow{*} q'$ is an abbreviation for the complement set of all other transitions leaving q. Moreover, such "*"-transitions can be implicit as TGV completes incomplete test purposes by a "*"-loop, thus allowing partial sequences in test purposes. TGV also supports regular expressions for the description of sets of labels. The left side of figure 5 gives an example of a test purpose TP for the specification S. Here we want to select sequences of actions which labels do not end with 5 or z (tau_5 and ! z) before a y followed by a z. "*"-loops are implicit in all states.

III. Principles and algorithms

This section describes the main algorithms of TGV. Let us sketch these algorithms, summarized in figure 4. TGV takes as inputs a specification S and a test purpose TP. The first operation performs a synchronous product between S and TP, marking S's behaviours accepted (or refused) by TP . From the result SP, we build the visible behaviour (traces and quiescence) in SP^{VIS}. Test selection then builds an IOLTS CTG by extraction of the accepted behaviors and inversion of inputs and outputs. Finally, all controllability conflicts are suppressed to conform with the definition of test cases. Alternatively, some conflicts can be suppressed during selection, leading to the construction of TG, and only residual conflicts are suppressed afterwards. When S is given implicitly by traversal functions, all operations except conflict resolution, can be applied on the fly, avoiding the construction of all complete IOLTS.

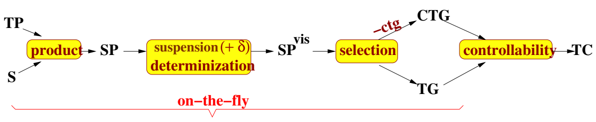

Fig. 4. Overview of test synthesis operations

A. Preliminary notions

A graph G with set of vertices V and set of edges E is denoted $G = (V, E)$. A *strongly connected component* (SCC) is a maximal subset V_i of V such that, for each pair (v_i, w_i) of vertices in V_i, there is a path from v_i to w_i and a path from w_i to v_i. An SCC is *trivial* if restricted to a single vertex with no loop. The partition of V into SCCs, defines a *reduced graph* which vertices are SCCs, and there is an edge from an SCC V_i to an SCC V_j if there is an edge in G from a vertex in V_i to a vertex in V_j.

In the sequel, we will see that several problems in test synthesis can be understood as reachability problems. Now, there is strong relation between reachability and SCC, as all vertices of an SCC have the same reachability properties: if a vertex w is reachable from a vertex u of an SCC V_i, w is reachable from all vertices in V_i.

Computation of SCCs: Tarjan [25] describes an algorithm of linear complexity for the computation of SCCs. In [19], we give an iterative version with "holes", and instantiate these "holes" for several algorithms used in TGV. The algorithm is a depth first traversal (DFS). Its principle is to identify SCCs by their *roots*, i.e. vertices first reached in the DFS. The DFS uses two stacks: the DFS-stack contains vertices of the current sequence and their pending edges, and the SCC-stack contains vertices which SCC is not completed. When an SCC root is popped from the DFS-stack, all vertices of the same SCC are on the top of the SCC-stack and are popped together.

B. Synchronous product

Test synthesis in TGV takes as inputs a specification S and a test purpose TP. The first problem is to mark S's behaviours accepted (on $Accept$ states) or refused (on $Refuse$ states) by TP. Just as in model-checking, this is solved by a synchronous product.

Definition 8: Let $S = (Q^{\mathrm{S}}, A^{\mathrm{S}}, \rightarrow_{\mathrm{S}}, q_0^{\mathrm{S}})$ be an IOLTS and $TP = (Q^{\mathrm{TP}}, A^{\mathrm{TP}}, \rightarrow_{\mathrm{TP}}, q_0^{\mathrm{TP}})$ a test purpose with $A^{\mathrm{TP}} = A^{\mathrm{S}}$ and equipped with state sets $Accept^{\mathrm{TP}}$ and $Refuse^{\mathrm{TP}}$.

The synchronous product $S \times TP$ is an IOLTS $SP = (Q^{\mathrm{SP}}, A^{\mathrm{SP}}, \rightarrow_{\mathrm{SP}}, q_0^{\mathrm{SP}})$, equipped with two disjoint sets of states $Accept^{\mathrm{SP}}$ and $Refuse^{\mathrm{SP}}$, and defined as follows. Its alphabet is $A^{\mathrm{SP}} \triangleq A^{\mathrm{S}} (= A^{\mathrm{TP}})$. Its state set Q^{SP} is the subset of $Q^{\mathrm{S}} \times Q^{\mathrm{TP}}$ reachable from the initial state $q_0^{\mathrm{SP}} \triangleq (q_0^{\mathrm{S}}, q_0^{\mathrm{TP}})$ by the transition relation $\rightarrow_{\mathrm{SP}}$ defined by: $(q^{\mathrm{S}}, q^{\mathrm{TP}}) \xrightarrow{a}_{\mathrm{SP}} (q'^{\mathrm{S}}, q'^{\mathrm{TP}}) \iff q^{\mathrm{S}} \xrightarrow{a}_{\mathrm{S}} q'^{\mathrm{S}} \wedge q^{\mathrm{TP}} \xrightarrow{a}_{\mathrm{TP}} q'^{\mathrm{TP}}$.
$Accept^{\mathrm{SP}} \triangleq Q^{\mathrm{SP}} \cap (Q^{\mathrm{S}} \times Accept^{\mathrm{TP}})$ and $Refuse^{\mathrm{SP}} \triangleq Q^{\mathrm{SP}} \cap (Q^{\mathrm{S}} \times Refuse^{\mathrm{TP}})$

The effect of the synchronous product is to mark behaviours of S by $Accept$ and $Refuse$, and possibly to unfold S. As TP is complete, all behaviours of S (including quiescence) are preserved in SP. SP is built by the next operation but could be built by any traversal.

C. Visible behaviours

The next operation consists in extracting the visible behaviour (traces and quiescence) from SP, i.e. constructing the IOLTS $SP^{\mathrm{VIS}} = (Q^{\mathrm{VIS}}, A^{\mathrm{VIS}}, \rightarrow_{\mathrm{VIS}}, q_0^{\mathrm{VIS}})$ such that $SP^{\mathrm{VIS}} = det(\delta(SP))$ (see definitions 3 and 2). SP^{VIS} is equipped with accept and refuse states:
$Refuse^{\mathrm{VIS}} = \{P \in Q^{\mathrm{VIS}} \mid P \cap Refuse^{\mathrm{SP}} \neq \emptyset\}$
$Accept^{\mathrm{VIS}} = \{P \in Q^{\mathrm{VIS}} \mid P \cap Accept^{\mathrm{SP}} \neq \emptyset\} \setminus Refuse^{\mathrm{VIS}}$.
which means that we choose to refuse a trace as soon as it corresponds to at least one refused sequence in SP. The right side of figure 5 gives the result of this computation for the examples S and TP where the exploration is stopped in Accept state 13 and Refuse states 5 and 7 (marked by loops labelled by Accept or Refuse).

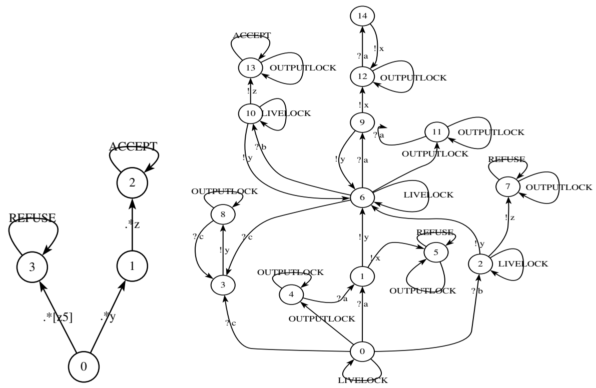

Fig. 5. Test purpose TP and $SP^{\mathrm{VIS}} = det(\delta(S \times TP))$

Computation of $det(\delta(.))$:

We already gave the definitions of δ and det, but for the sake of efficiency, quiescence and determinization are computed simultaneously. Figure 1 will serve to illustrate the computation for the example S even if we apply it to $S \times TP$.

Theoretically, a δ loop should be added in each quiescent state. For deadlocks (no deadlock in S) and output quiescent states (states 1,7,9), we just look at outgoing transitions. For livelocks, which are loops of internal actions (in states 0-2 and 4), a δ loop should be added in each state of a non trivial SCC of internal actions (τ-SCC for short). But, as $\delta(S)$ is determinized afterwards, adding a δ loop in the root of each τ-SCC as the same effect on $\Rightarrow$. We will see how to factorize this with determinization.

Determinization: Determinization consists in building $det(S)$ starting from its initial meta-state $q_0^{\mathrm{S}} = q_0^{\delta(\mathrm{S})}$ after ϵ (in the example, the meta-state 0 of $det(\delta(S))$ is $\{0, 1, 2, 9\}$) by alternation of two operations:

- subset construction: for a state set P and a visible action a, compute the set $P' = \{q' | \exists q \in P, q \xrightarrow{a} q'\}$, of states reachable in one visible step a from P. For $P = \{0, 1, 2, 9\}$ and $?a$, $P' = \{3, 8\}$.
- ε-closure: for a state set P, compute the set P after ε of states reachable from P by sequences of internal actions. For $P = \{3, 8\}$ P after ε is also $\{3, 8\}$.

In [18] we propose an ε-closure algorithm that avoids redundancies, with the counterpart of a supplementary memory complexity. The idea is as follows. For all states q of a τ-SCC, $Fire(q) =$

$\{(a,q')|a \in A^{\delta(\text{SP})} \setminus I^{\delta(\text{SP})}, q' \in Q^{\delta(\text{SP})}, q \overset{a}{\Rightarrow} q'\}$ is identical. In fact $Fire(q)$ denotes visible actions after a τ sequence and resulting states, thus a reachability property. For example $Fire(0) = Fire(2) = \{(?a,3),(?a,8),(?b,4),(?c,6)\}$ and $Fire(1) = \{(?a,3)\}$. A meta-state ($\{0,1,2,9\}$ for example) is not only a set of states but a reduced graph of τ-SCC ($\{0,2\}$, $\{1\}$, $\{9\}$) and $Fire(racine(V_i))$ is synthesized on each τ-SCC V_i. Meanwhile, quiescence is computed and δ-loops added. In particular a livelock is a non trivial τ-SCC ($\{0,2\}$). Then, when an already visited state q is reached by a new call to ε-closure, the root of its τ-SCC returns $Fire(q)$. For a meta-state P, the set $Fire(P) = \bigcup_{V_i \in SCC\_init} Fire(racine(V_i))$ where SCC_{init} is the set of initial SCC of the reduced graph of τ-SCC of P gives all fireable transitions and reached states. Thus it gives the result of the subset construction. The time complexity of determinization remains exponential but, by avoiding redundancy, our algorithm is much more efficient than the naïve one.

A word on minimization: The IOLTS SP^{VIS} built is not minimal w.r.t. trace equivalence. As partition refinement algorithms used for minimization work backward, they need the complete IOLTS. But on-the-fly test synthesis (see section III-F) avoids the complete construction of SP^{VIS} and works forward. We then use a weaker equivalence relation and minimize SP^{VIS} on-the-fly for this relation: two meta-states P_i and P_j of SP^{VIS} are "1-step equivalents" if $Fire(P_i) = Fire(P_j)$. This minimization is simply done by a coding of each meta-state P_i by $Fire(P_i)$ which is the only used information in P_i.

D. Test selection

SP^{VIS} represents all visible behaviours of S and, among these, those behaviours accepted (or refused) by the test purpose TP with the sets $Accept^{\text{VIS}}$ and $Refuse^{\text{VIS}}$. The next operation consists in extracting a test case by selection of accepted behaviours. This operation is a bit more complex as, to compute a test case (see definition 5), we must perform a mirror image (invert inputs and outputs), complete it for inputs in all states where an input is possible, ensure controllability, and define verdicts by sets **Pass**, **Inconc** and **Fail**.

In a first step, we will not deal with controllability, and will describe the computation of an IOLTS CTG for *Complete Test Graph*. CTG is an interesting IOLTS as it contains all test cases corresponding to the test purpose. Moreover, it is easier to explain separately how controllability conflicts are solved.

Definition 9: For a specification S and a test purpose TP, the *complete test graph* is an IOLTS $CTG = (Q^{\text{CTG}}, A^{\text{CTG}}, \longrightarrow_{\text{CTG}}, q_0^{\text{CTG}})$, with three sets of trap states **Pass**, **Inconc** and **Fail**, and defined from $SP^{\text{VIS}} = det(\delta(S \times TP))$ as follows:
The alphabet is $A^{\text{CTG}} = A_{\text{O}}^{\text{CTG}} \cup A_{\text{I}}^{\text{CTG}}$ with $A_{\text{O}}^{\text{CTG}} \subseteq A_{\text{I}}^{\text{VIS}}$ and $A_{\text{I}}^{\text{CTG}} = A_{\text{O}}^{\text{VIS}}$ (mirror image), The set of states is $Q^{\text{CTG}} = L2A \cup$ **Inconc** $\cup$ **Fail**, with $L2A = \{q \in Q^{\text{VIS}} \mid \exists \sigma \in A^{\text{VIS}^*}, v \xrightarrow{\sigma}_{\text{VIS}} Accept^{\text{VIS}}\}$, (L2A for *leads to Accept*) i.e. the set of states from which $Accept^{\text{VIS}}$ is reachable, and $Inconc = \{v \in Q^{\text{VIS}} \mid \exists u \in L2A, v \notin L2A,\ a \in A_{\text{O}}^{\text{VIS}},\ u \xrightarrow{a}_{\text{VIS}} v\}$, i.e. states not in $L2A$ but direct successors of states in $L2A$ by an output in SP^{VIS}. **Fail** $= \{Fail\}$ where $Fail \notin Q^{\text{VIS}}$ is a new state. If $q_0^{\text{VIS}} \in L2A$, the initial state is $q_0^{\text{CTG}} = q_0^{\text{VIS}}$ and Q^{CTG} is restricted to states reachable from q_0^{CTG} by $\rightarrow_{\text{CTG}}$, otherwise Q^{CTG} is empty. The transition relation is $\rightarrow_{\text{CTG}} = \rightarrow_{\text{L2A}} \cup \rightarrow_{\textbf{Inconc}} \cup \rightarrow_{\text{Fail}}$ where $\rightarrow_{\text{L2A}} = \rightarrow_{\text{VIS}} \cap (L2A \times A^{\text{CTG}} \times L2A)$, $\rightarrow_{\textbf{Inconc}} = \rightarrow_{\text{VIS}} \cap (L2A \times A_{\text{I}}^{\text{CTG}} \times \textbf{Inconc})$, and $\rightarrow_{\textbf{Fail}} = \{(v, a, Fail) \mid v \in L2A \wedge a \in A_{\text{I}}^{\text{CTG}} \wedge v \not\xrightarrow{a}_{\text{VIS}}\}$.
Finally, **Pass** $= Accept^{\text{VIS}}$.

The left side of figure 6 illustrates the computation of the complete test graph from SP^{VIS} for the examples S and TP. In SP^{VIS}, the SCC $\{0\}$, $\{1\}$, $\{2\}$, $\{4\}$, $\{6,9,10,11\}$ and $\{13\}$ lead to Accept, thus their states and transitions are preserved in CTG. $\{3,8\}$ does not lead to Accept and is cut as it is reached by the input $?c$ but outputs $!x$ and $!z$ leading to $\{5\}$ and $\{7\}$ are preserved and lead to an Inconclusive verdict.

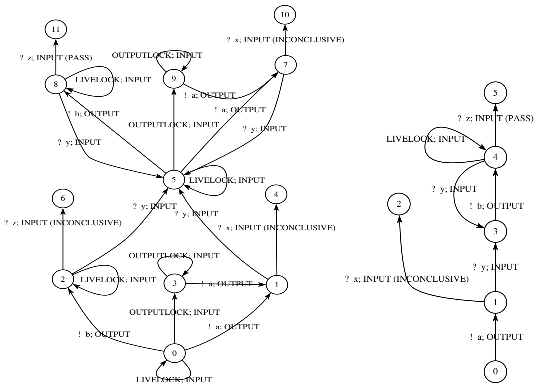

Fig. 6. Complete test graph and test case

Algorithm: According to the definition of CTG, the main point is to compute the set $L2A$ and to check if $q_0^{\text{VIS}} \in L2A$. Now, the set $L2A$ consists of states where the CTL [5] property $L2A = EF Accept^{\text{VIS}}$ holds. This is clearly a reachablity property. Thus, either all states of an SCC are in $L2A$ or none of them. The algorithm, called TGVloop adapts Tarjan's algorithm by the additional synthesis of the attribute $L2A$ and a construction of $\rightarrow_{\text{L2A}}$ during backtracking. This algorithm can be seen as a model-checking algorithm for $L2A$ producing all witnesses of $L2A$ starting in the initial state. Moreover, the computation of **Inconc** and $\rightarrow_{\textbf{Inconc}}$ is done during backtracking of output transitions of SP^{VIS} from states in $L2A$ to states outside $L2A$. The *Fail* state and transitions in $\rightarrow_{\textbf{Fail}}$ are implicit, $\rightarrow_{fail}$ being defined by complementation of fireable transitions. The algorithm has linear complexity in time and space, just like Tarjan's SCC algorithm.

E. Pruning controllability conflicts

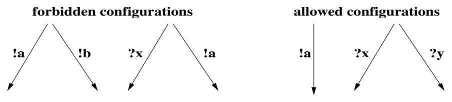

Fig. 7. Controllability conflicts

CTG satisfies all properties required for a test case (definition 5), except controllability: some states q of CTG may have a choice between outputs or between inputs and outputs. (see figure 7). Solving these conflicts consists in extracting a controllable subgraph of CTG while preserving other required properties. In a state with a conflict, some transitions must be pruned: either one output is kept and all other outputs and inputs are pruned, or all inputs are kept and outputs are pruned. Unreachable states are suppressed. Reachability to $Accept$ (or **Pass**, synthesized in $L2A$) is preserved by a backward traversal of CTG from **Pass** states to the initial state. Among possible traversal strategies we choose a breadth first traversal for its ability to select shorter paths from **Pass**. The right side of figure 6 shows a test case which is one possible result of this conflict resolution. In the state 0 of CTG, $!a$ is chosen and $!b$ and the δ inputs are pruned, and in state 5, $!b$ is chosen and $?a$ and δ inputs are pruned. Note that $!b$ could be chosen in 0 but $!a$ cannot in 5 as the PASS verdict would be unreachable.

Forward pruning: Conflict resolution with a backward traversal, as presented previously, requires a complete construction of CTG. But this reduces the interest of on-the-fly synthesis (see section III-F). Another solution consists in pruning during backtracking in TGVloop. This may solve some controllability conflicts and avoid the construction of some parts of CTG. But some conflicts may not be solved this way, only in the case of particular traversal orders in loops. However, residual conflicts can be solved *a posteriori* by the backward algorithm. In the example, the conflict in 0 can be solved by forward pruning but the conflict in 5 is solved this way only if $!b$ is explored before $!a$..

F. On the fly test case synthesis

Figure 4 gives an overview of the operations needed for test synthesis. After the computation of the product $SP = S \times TP$, the suspension automaton $\delta(SP)$ and the deterministic automaton of it $SP^{\text{VIS}} = det(\delta(SP))$ are factorized in one operation. $Accept$ and $Refuse$ sets are propagated by these operations. Then, either a complete test graph CTG is computed by selection of traces leading to $Accept^{\text{VIS}}$, mirror image, addition of verdicts, or a test graph TG is build by pruning some controllability conflicts during selection. Finally, residual controllability conflicts on CTG or TG are solved to produce one test case TC.

In general TC is small compared to S, because of selection by TP. Also the specification is not given explicitly by an IOLTS but in a specification language. Its semantics is an IOLTS S, but it is given implicitly by a simulator API in terms of functions allowing its traversal. Building S completely when only a small part is used in TP is thus inefficient, and in general impossible if S is not finite state.

The idea of on-the-fly synthesis is a lazy construction of subgraphs of S, SP and SP^{VIS} necessary for the construction of TC, i.e. selected by TP. To understand the global behaviour, one has to reason in terms of functions for the construction of each of the IOLTS S, SP and SP^{VIS}. The required functions are traversal functions (*init* giving the initial state, *fireable* which gives the set of fireable transitions in a state, *succ* which, from a state and a fireable transitions, computes the target state(s)), a comparison function, and functions computing the membership to $Accept$ or $Refuse$.

In the worst case, on-the-fly synthesis does not reduce the construction of the IOLTS S, SP and SP^{VIS}. But in practice, the reduction is often dramatic, in particular if TP constraints the behaviours by the use of $Refuse$ states. Using this technique often allowed us to quickly synthesize test cases on very large or even infinite state spaces. Nevertheless, it is clear that if S is small, it is preferable to build it completely and to minimize it before test synthesis with different test purposes. As we already noticed, on-the-fly test synthesis does not allow minimization for trace equivalence, and this sometimes results in the unfolding of loops in test cases. However, as test cases are often small, they can be minimized *a posteriori*.

IV. The TGV tool

TGV architecture: it follows the functional description (see figure 8). TGV is made of software levels communicating through API. Each level implements one of the algorithms described in section III and transforms an IOLTS (or two in the case of the product) given by its simulation API, into a simulation API of a new IOLTS. Additionally, TGV uses libraries for storing states, for hiding, renaming and regular expressions of the CADP toolbox [11]. Due to this architecture, TGV allows to guide simulation API of different specification languages with the same source code, except for highest API. This ensures the coherency of different variants, and facilitates the porting on new systems (TGV works on SunOS 5, Linux and WindowsNT). Moreover some parts can be used alone, or by other programs. In particular, we have implemented a module called VTS which verifies soundness and laxity of manual test cases. This module just replaces TGVloop and uses other levels. It also served for testing TGVloop.

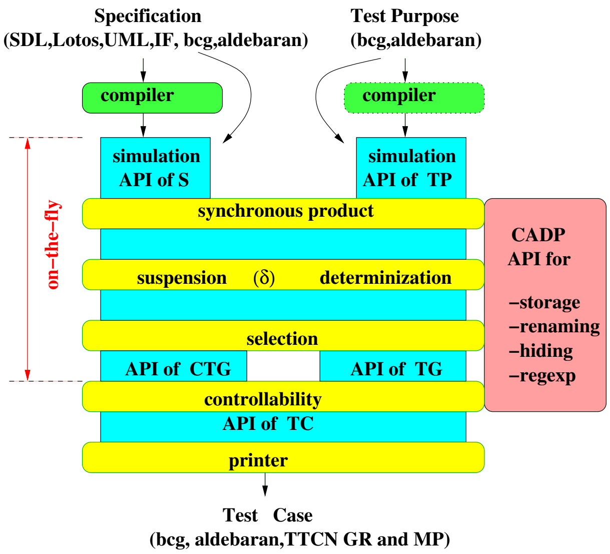

Fig. 8. TGV architecture

A. Supported languages

TGV supports different specification language by a connection to their simulation API:

Lotos: TGV uses the simulation API provided by the CAESAR compiler of the CADP toolbox. But as Lotos does not distinguish inputs and outputs, TGV needs an additional file which partitions visible events into inputs and outputs.

SDL: TGV uses the simulation API of the ObjectGéode SDL tool (Telelogic) [13]. There exist two versions of this connection. The academic version uses a CADP-like API and pilots the ObjectGéode simulator. The commercial tool TestComposer of ObjectGéode also integrates TGV as one of its two test synthesis engines. TestComposer is also equipped with a test purpose synthesis engine based on a branch coverage strategy. This engine produces sequences of observable actions interpreted as test purposes.

UML: to produce test cases from UML models, TGV is connected to a CADP-like simulation API provided by UMLAUT [14], a validation framework for UML developed in IRISA. UMLAUT uses class and objet diagrams, deployment diagrams, state diagrams and gives an operational semantics to UML by transformation and compilation of the UML model.

IF: IF is an intermediate form developed by Verimag (Grenoble) that can be produced from SDL and UML. TGV also uses the simulation API provided by the IF compiler.

Your favorite specification language: the simulation API required by TGV is documented and quite simple. For a language with an operational semantics in terms of LTS or IOLTS, if a compiler produces a simulation API, an interface between this API and the TGV API can be easily built.

As a result, TGV may produce test cases in TTCN (Tree and Tabular Combined Notation [1]), or in one of the graph formats (.aut and .bcg) of CADP.

B. Other TGV characteristics

Several options are provided by TGV. In particular, TGV produces test cases with timer operations. Two timers are managed, TAC and TNOAC. TAC is started when inputs are expected (except if δ is expected). If an input is observed, TAC is cancelled, otherwise a timeout is observed and produces a *Fail*verdict. TNOAC is started before entering a state where a quiescence (δ) is allowed, it is cancelled if an input is observed and the observation of δ is replaced by a timeout that does not produce a *Fail*verdict, as it is specified.

The traversal depth can be bounded. This bound is interpreted in terms of visible actions as, due to nondeterminism, a bound in terms of actions could result in an unsound test case.

TGV allows the computation of postambles from PASS and INCONC verdicts. If possible, these postambles lead to stable states i.e. states where, according to S, no output from the IUT is expected.

C. Case studies

Different versions of TGV have been experimented on industrial size case studies, in various application domains, and with different specification languages. We just sketch these cases studies, as they have been already published.

SDL: The DREX protocol, a military version of the ISDN D protocol, allowed the validation of TGV principles on a preliminary version of TGV. This version was incomplete and not on-the-fly [12]. The experiment allowed to compare TGV with the TVéda tool from CNET [23] and the TOPIC prototype from Vérilog, as well as the manual production of test cases [8]. Another one, the SSCOP protocol has been used in several experiments with variations on the number of PCO, the communication mode (synchronous or asynchronous) between tester and IUT, with the aim of putting into relief the particularities of TGV [3]. Also a part of the TTCN test suite produced by ATM Forum has been checked with VTS (for soundness and laxity) [17] and some errors due to asynchronism were detected.

Lotos: Several experiments [20], [21] consisted in using TGV on specifications of a multiprocessor architecture of Bull. Produced test cases have been executed by Bull on a simulator of the architecture. TGV has also be used on a conference protocol [9]. Test cases have been executed on 28 mutants of a correct implementation in order to compare TGV with the TorX tool from Twente. Both tools detected all incorrect mutants. A problem with TGV was to imagine adequate test purposes.

UML: A model of an air traffic controller is used as an example of the UMLAUT/TGV connection.

IF: In the framework of the IST European project Agedis, TGV has been used on an IF specification (translated from an SDL model designed from an UML one) of a component of the Transit Computerization Project. The number of processes (10) and their concurrency pushed TGV to its limits and gave us some ideas about possible improvements (see section V).

D. Comparison with other techniques and tools

TGV can be compared with test synthesis techniques and tools based on model-checking (e.g [10]. The common idea of most of these techniques is to use a standard model-checker to produce counter-examples of the negation of a property (interpreted as a test purpose), abstract these sequence from internal actions and interprete them as test cases. But TGV goes beyond by the use of a clear testing theory and a real adaptation of model-checking algorithms to test synthesis, which allows to take into account non-deterministic and non controllable specifications.

The most comparable tool is TorX [7], the tool from the University of Twente. The testing theory is almost identical (except that livelocks are not considered). It also synthesizes test cases on the fly, but for the moment without any test purpose. As it executes test cases on the fly during their synthesis, the test case synthesis is guided by the observations made on the IUT on the proposed stimuli. As mentionned in subsection IV-B both tools where applied to the same case study and despite their differences, gave similar results in terms of detection power.

V. Conclusion and perspectives

In this paper, we have presented the principles of TGV, its underlying theory, the algorithms and the tool. TGV has improved the state of the art in test synthesis in a significant way. Our main contribution is not in the theory despite our adaptations and improvements, but in the algorithms and tool architecture. TGV is able to synthesize tests from industrial size specifications. However, some improvements are still necessary for an industrial use.

A first drawback is the necessity to describe test purposes. It is an advantage compared to manual generation of test cases because test purposes are of an higher abstraction level and TGV ensures soundness of synthesized test cases. But their is an effort to be paid for the description of test purpose and this requires some expertise. TestComposer provides a partial answer by the synthesis of test purposes for a coverage criteria. But the branch coverage criteria is often too weak and some test purposes still have to be written. A possible direction for future research is to use improved coverage criteria based on the specification code and adapted to the specific problem of conformance.

Improvements of algorithms are investigated. An interesting direction is to use partial order techniques as in model-checking [22]. These techniques can already be used for internal actions as the order of occurrence of internal actions has no effect (if they are not used in test purposes) on visible actions, thus on synthesized test cases. Applying these techniques for visible actions is more difficult as concurrent behavior must be synthesized in test cases. Other improvements concern compositionality. We investigate how to compute test cases incrementally in the case of compositional specifications and, in the context of Agedis, how to compute several test cases in one run from a composition of test purposes or coverage criteria.

Another important problem is the problem of distributed testing. In the general case, the system is distributed and test cases should be distributed and should communicate asynchronously. Concurrent-TTCN gives such a specification power. A first approach we adopted [16] is to synthesize a sequential test case and to distribute it according to localities of actions. Global choices were solved by a consensus. The main drawback is the lost of concurrency and the fact that unnecessary synchronizations between testers are added. A direction of research is to preserve concurrency by the use of true concurrency models [15] and to revisit the testing theory accordingly.

Another drawback of TGV is the use of enumerative techniques. A consequence is that specifications with data structures with large (or infinite) domains may be impossible to treat, even with on-the-fly techniques. Parametric specifications are out of the scope of TGV. A solution is to use symbolic techniques [24]. States sets and transitions are not enumerated but represented by predicates. The specification model we use is called IOSTS (Input-Output Symbolic Transition Systems). Transitions are labelled with inputs, outputs or internal actions, guarded with boolean expressions on variables and parameters and communication variables, and may perform assignments. From an IOSTS test purpose (with Accept and Refuse states) and a specification in the form of IOSTS, a test case is extracted with techniques similar to TGV, but only on the syntax of the specification. This test case is sound for the conformance relation but may include unsatisfiable transitions that should be pruned. In its current status, our tool STG [4] only allows to prune some locally unsatisfiable transitions with the Omega constraint solver. A deeper analysis (using static analysis, abstraction, proof) could improve the tool. Nevertheless, executable test cases can be produced and executed on implementations, which requires to fix the values of parameters. During execution, Omega is also used to find outputs satisfying the guards.

Acknowledgements: The TGV tool is the result of a common work during several years. We wish to thank all participants in its design and development in Irisa and Verimag: Jean-Claude Fernandez, Alain Kerbrat, Pierre Morel, Laurence Nedelka, Joseph Sifakis, Séverine Simon, César Viho and students. We also thank Laurent Mounier and Marius Bozga from Verimag for the connection to IF, Hubert Garavel and the VASY team from Inria Rhônes-Alpes for their help in the connection of TGV with Lotos, and the support and distribution of TGV in the CADP toolbox.

REFERENCES

[1] I. 9646. 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.

[2] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53(3), 1987.

[3] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, 36(1):27–52, Jan. 2000.

[4] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *e-Smart 2001, International Conference on Research in Smart Cards*, 2001. to appear in LNCS.

[5] E. Clarke and E. A. Emerson. Synthesis of synchronisation skeletons for branching time temporal logic. In *Workshop in Logic of Programs, (Yorktown Heights, NY)*, volume 131 of *LNCS*. Springer Verlag, 1981.

[6] R. De Nicola and M. Henessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[7] R. G. De Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[8] L. Doldi, V. Encontre, J.-C. Fernandez, T. Jéron, S. Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten and A. Burkhardt, H.-J.Giessler, editors, *IFIP TC6 9^{th} International Workshop on Testing of Communicating Systems*. Chapman & Hall, Sept. 1996.

[9] L. Du Bousquet, S. Ramangalahy, S. Simon, V. C., A. Belinfante, and R. G. De Vries. Formal test automation: The conference protocol with tgv/torx. In H. Ural, R. Probert, and G. v. Bochmann, editors, *IFIP 13^{th} Int. Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers, 2000.

[10] A. Engels, L. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model-Checking. In *Third Workshop TACAS, Enschede, The Netherlands*, LNCS 1217. Springer-Verlag, 1997.

[11] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. of CAV'96 (New Brunswick, New Jersey, USA)*. LNCS 1102, August 1996.

[12] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of conformance test suites for protocols with verification technology. *Science of Computer Programming*, 29:123–146, 1997. Egalement disponible en rapport de recherche Irisa n^o 1035 et Inria n^o 2923.

[13] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999.

[14] W.-M. Ho, J.-M. Jézéquel, A. L. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, Oct. 1999.

[15] C. Jard. Principles of test synthesis using true-concurrency models. In H. König and I. Schiferdecker, editors, *Proc. of Testcom'2002*, Berlin, Germany, March 2002. IFIP.

[16] C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards automatic distribution of testers for distributed conformance testing. In *FORTE/PSTV'98, Paris, France*. Chapman & Hall, Nov. 1998.

[17] C. Jard, T. Jéron, and P. Morel. Verification of test suites. In *TestCom 2000, IFIP TC 6 / WG 6.1, The IFIP 13th International Conference on Testing of Communicating Systems, Ottawa, Ontario, Canada*. Kluwer Academic Publishers, Aug. 2000.

[18] T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée: application à la génération de test. In *CFIP'97, Congrès Francophone sur l'Ingéniérie des Protocoles, Liège, Belgique*. Hermes, Sept. 1997.

[19] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 108–122. Springer-Verlag, July 1999.

[20] H. Kahlouche, C. Viho, and M. Zendri. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11^{th} International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.

[21] H. Kahlouche, C. Viho, and M. Zendri. Hardware Testing using a Communication Protocol Conformance Testing Tool. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, March 1999.

[22] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *CAV Workshop*, volume 818 of *LNCS*. Springer Verlag, 1994.

[23] M. Phalippou. Test Sequence Generation Using Estelle or SDL Structure Information. In *FORTE'94*, Berne, October 1994.

[24] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Integrated Formal Methods (IFM'00), Dagstuhl, Allemagne*, volume 1945 of *LNCS*, pages 338–357. Springer Verlag, Novembre 2000.

[25] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[26] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17, 1996.

[27] I. J. WG7. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500. ISO - ITU-T, itu-t sg 10/q.8, 1996.