Int J Softw Tools Technol Transfer (2005) 7: 297–315 / Digital Object Identifier (DOI) 10.1007/s10009-004-0153-x

TGV: theory, principles and algorithms

A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems

Claude Jard[1], **Thierry Jéron**[2]

[1] IRISA/ENS Cachan-Bretagne, Campus de Ker Lann, F-35170 Bruz Cedex, France
e-mail: Claude.Jard@bretagne.ens-cachan.fr
[2] IRISA/INRIA, Campus de Beaulieu, F-35042 Rennes Cedex, France
e-mail: jeron@irisa.fr

Published online: 20 October 2004 – © Springer-Verlag 2004

Abstract. This paper presents the TGV tool, which allows for the automatic synthesis of conformance test cases from a formal specification of a (non-deterministic) reactive system. TGV was developed 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, which is based on a model of transitions system which distinguishes inputs, outputs and internal actions, and is based on the concept of conformance relation. The principles of the test synthesis process, as well as the main algorithms, are explained. We then describe the main characteristics of the TGV tool and refer to some industrial experiments that have been conducted to validate the approach. As a conclusion, we describe some ongoing work on test synthesis.

Keywords: Conformance testing – Test generation/synthesis – Reactive systems – Protocols – Model-checking – Transition systems

1 Conformance testing

Software systems have becoming increasingly complex, distributed and reactive. Their reliability is a major concern, in particular for critical systems as errors occurring during their execution may have dramatic economic or human consequences. Correctness is also essential for less critical software. It is thus essential to try to preserve the correctness of software all along the design process until deployment. This includes software engineering methods, verification and validation. But full correctness is in general impossible to prove. Testing is then one of the most popular validation techniques. Testing aims at discovering bugs in design or implementation phases with respect to a reference. It cannot prove correctness, but it improves confidence. Testing may focus on different aspects of software such as functionality, robustness, performance, timing constraints etc. It may be employed at different levels, from unit testing to integration and system testing. One of the main problems of testing is choosing test data. This choice may be based on the code (white box testing) or on the specification (black box testing), depending on the availability and complexity of these artefacts. In practice, testing most often remains a craft activity. Test data are selected arbitrarily, and test execution and test results analysis are performed manually. This implies that testing is very costly. However, most phases can be automated, at least partially.

In this paper, we focus on *conformance testing* applied to non-deterministic reactive systems. By reactive system we mean a software component which reacts to the stimuli of its environment. Non-determinism means that different reactions can be obtained after applying a given stimulus (this is typically the case in the presence of concurrency in systems). Conformance testing consists of checking that the behaviour of a real implementation of a system (IUT for *implementation under test*) is correct with respect to a specification. The code of the IUT is unknown, and its behaviour is only visible by interaction with a tester. This 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. In this context, we will show how automation can significantly improve test selection.

1.1 Some basic concepts

In the context of telecommunication protocols, the main concepts of this activity are described in the standardization document ISO 9646 [19]. Some of them are introduced here.

A *test case* is an elementary test targeted at testing a particular functionality, called *a test purpose*. A *test* *suite* is a set of test cases. The basic elements of a test case are interactions through PCOs: outputs are stimuli sent from the environment in order to control the IUT's input events; inputs are observations of the IUT's outputs to the environment. Inputs may lead to different *verdicts*. A Fail verdict denotes a divergence from the expected behaviour: the IUT is rejected. A Pass verdict is returned if the observation is correct and the test purpose is reached. Sometimes, one wants to bring the IUT into a particular state or the initial state after a test case by a procedure called *postamble*. In this case, a non-definitive (Pass) first verdict is returned and the Pass verdict is only returned if the postamble does not detect any non-conformance. An Inconclusive verdict is returned if correct behaviour 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: they may have a choice between several output interactions to the same input. The *tester* – specialized hardware, software or human operator – executes test cases. But as test cases are often described at some abstraction level (they are called *abstract test cases*), they must be translated into *executable test cases* (level on which all the coding aspects of data and interactions have been resolved).

1.2 Formalizing for automation

Conformance testing is a costly activity which plays an important part in the global cost of software. For a long time the scientific community has tried to automate the process of deriving test cases. For conformance testing, the reference behaviour is described by a specification which determines the verdicts: it plays the role of an *oracle*, as it is called in the general framework of testing. Automation thus requires formalizing the specification, but also formalizing the interaction between the tester and the IUT. The definition of verdicts also forces formalizing conformance, i.e. the relation between the IUT and its specification that is checked during testing. Algorithms for automatic test case synthesis must be designed that take specifications as inputs. Essential properties of test cases must be established. *Soundness* means that test cases may only reject non-conformant IUTs, *exhaustiveness* means that all non-conformant implementations are rejected by a test suite (or may be rejected). The main ingredients for automation are described in [20].

Several approaches to conformance test generation have been studied. For protocol testing, two approaches have been studied, initially focused on control. A first approach uses finite-state machines (FSM) as a specification model (see [29] for a survey or [31] for an annotated bibliography). The principle of testing is to check that an unknown FSM, the IUT, is equivalent to the specification. A finite set of finite test sequences is generated which proves or disproves this equivalence. Of course, this is possible only if the set of possible IUTs is finite, which is ensured by hypothesis on both the specification and the IUT. Another approach was initially based on labelled transition models and testing preorders [1, 8] and further improved by distinguishing inputs and outputs [35]. Hypotheses on specifications and IUTs are weaker. In particular, non-deterministic specifications (in the sense of automata) can be handled. The counterpart is that a finite set of test cases cannot prove conformance.

A third approach to formal conformance testing was initially focused on data. It is based on algebraic data types [3]. The principle is to test axioms, test cases being terms of the algebra. Starting from an (infinite) exhaustive test set, hypotheses such as regularity or uniformity are added to restrict the size of the exhaustive test set to a finite one. This approach has been extended with LTS methods for Lotos specifications mixing control and data [16].

The tool TGV presented in this paper uses the approach based on labelled transition models. This means that it is based on behavioural models of specifications in terms of labelled transition systems. This does not exclude data in specifications, but it means that data values are enumerated in the model. TGV is also based on a precise testing theory which allows us to describe test generation algorithms and establish important properties on generated test cases. This is essential for us if we are to gain confidence in software by testing. Nevertheless, the TGV approach is not only a theoretical work; it is also an efficient tool that has proved useful in numerous case studies. This efficiency is mainly due to the on-the-fly approach which allows us to generate test cases by a partial exploration of state graphs, thus avoiding the state explosion problem.

The paper presents the entire TGV approach and is organized as follows. In Sect. 2 we briefly describe the functional view of TGV. In Sect. 3 we present the underlying testing theory of TGV based on the model of labelled transition systems with distinguished inputs and outputs and precise notions of conformance and verdicts. Then Sect. 4 presents the synthesis algorithms and the properties that can be established on generated test cases. The TGV tool is described in Sect. 5. Some case studies are described in Sect. 6, ending with lessons learned from these case studies. In Sect. 7 we compare the TGV approach with other techniques and tools. Finally, we conclude and present some perspectives in Sect. 8.

2 TGV functional view

TGV is a tool for test generation from specifications. Its functional view is sketched in Fig. 1. One of the inputs is thus a specification of the intended behaviour of the system under test. As will be seen later in Sect. 5, TGV is not dependent on any particular specification language, but rather depends on a particular semantics of these languages. This semantics should focus on the behaviour.

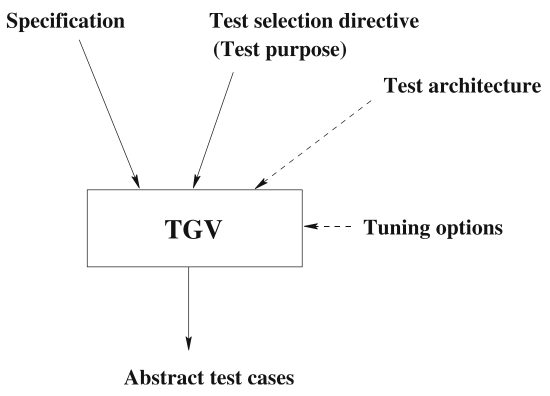

Fig. 1. Functional view of TGV

The semantics of the specification thus describes valid behaviours of the system under test.

TGV's role is to select test cases from the behaviours of the specification. For this, one needs to give a second input that can be called a test selection directive. Test selection directives could take different forms, including random test selection, selection guided by coverage criteria, guided by test purposes, or a mixture of these. Even if TGV now allows different selection directives (see the IF paragraph of Sect. 5.2), selection in TGV was originally based on test purposes. Test purposes are specified by automata that accept behaviours of the specification. They allow us to describe targeted behaviours in an abstract way as it is not necessary to describe complete sequences of actions. More general test selection directives have recently been added to TGV, mixing extended test purposes with test coverage directives, but in this paper we will mainly focus on simple test purposes.

Other optional inputs can be given to TGV. First, inputs are used to refine the test architecture from the default one implicitly defined by the specification. Second, options can be given to tune the test selection algorithms.

In its original form, TGV generates abstract test cases from a specification and a test purpose. Abstract test cases describe behaviours in terms of input/output interactions between the tester and the IUT, and verdicts associated with those behaviours. Abstract test cases produced by TGV are in a generic format of graphs. These graphs can be easily translated into a specific language for the description of abstract test cases such as in TTCN.

Abstract test cases are not directly executable on an IUT, but specialized tools allow us to transform these abstract test cases into executable test cases. TGV does not take this phase into account.

3 Testing theory in TGV

The contribution of TGV to automatic synthesis of test cases is mainly in the area of algorithms and tools. TGV is based on a conformance testing theory, inspired by the work of Jan Tretmans and his colleagues (at the University of Twente) [35]. This theory builds on previous work on testing equivalences and preorders [1, 8]. The behaviours of specifications and IUTs 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 by non-specialists.

3.1 Modelling with transition systems

Labelled transition systems (LTS) have long been used to define the semantics of behavioural specifications. LTSs are represented by graphs whose states represent configurations of systems, and edges represent moves between these configurations on the occurrence of actions. Usually LTSs make a difference between internal and visible actions. But for conformance testing, a distinction must also be made between events of the system that are *controllable* by the environment (the inputs) and 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* $M = (Q^M, A^M, \to_M, q_0^M)$*, where* Q^M *is a finite non-empty set of states,* $q_0^M \in Q^M$ *is the initial state and* A^M *is the alphabet of actions. It is partitioned into three sets* $A^M = A_I^M \cup A_O^M \cup I^M$. A_I^M *is the input alphabet,* A_O^M *is the output alphabet and* I^M *the alphabet of internal actions.* $\to_M \subseteq Q^M \times A^M \times Q^M$ *is the transition relation.*

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

Notations: Let $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, \to_{\mathrm{M}}, q_0^{\mathrm{M}})$ be an IOLTS. The subscript (or superscript) M will be omitted when clear from the context. We write $q \xrightarrow{a}_{\mathrm{M}} q'$ for $(q, a, q') \in \to_{\mathrm{M}}$ 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 $M \to_{\mathrm{M}}$ for $q_0^{\mathrm{M}} \to_{\mathrm{M}}$. Let $\mu_{(i)} \in A^{\mathrm{M}}$ be 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 and $q, q' \in Q^{\mathrm{M}}$ some states.
$\Gamma(q) \triangleq \{\mu \in A^{\mathrm{M}} \mid q \xrightarrow{\mu}_{\mathrm{M}}\}$ is the set of firable actions in q.
$Out_{\mathrm{M}}(q) \triangleq \Gamma(q) \cap A_{\mathrm{O}}^{\mathrm{M}}$ is the set of firable outputs in q; we extend it to sets of states: for $P \subseteq Q^{\mathrm{M}}$
$Out_{\mathrm{M}}(P) \triangleq \{Out_{\mathrm{M}}(q) \mid q \in P\}$.

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

Visible behaviours are described by the $\Rightarrow$ relation. We define $q \overset{\varepsilon}{\Rightarrow} q' \triangleq q = q'$ or $q \xrightarrow{\tau_1 \cdot \tau_2 \cdots \tau_n}{}^{*} q'$ 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* $\sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$ (respectively P *after* $\sigma \triangleq \bigcup_{q \in P} q$ *after* σ) is the set of states reachable from q (respectively from the state set P) by action sequences from which only the projection σ

onto visible actions is defined. $Traces(q) \triangleq \{\sigma \in (A \setminus I)^* \mid q \overset{\sigma}{\Rightarrow}\}$ (respectively $Traces(M) \triangleq Traces(q_0^{\mathrm{M}})$) describes the sequences of visible actions firable from q (respectively from the initial state of an IOLTS M).

From an IOLTS M, it is possible to build a deterministic IOLTS with the same traces as M. This IOLTS represents the visible behaviour of M.

Definition 2. *Let $M = (Q^M, A^M, \to_M, q_0^M)$ be an IOLTS. The deterministic IOLTS of M, denoted by $det(M)$, is a deterministic IOLTS defined by*

$$det(M) = (2^{Q^M}, A^M \setminus I^M, \to_{det}, q_0^M \text{ } after\, \varepsilon) \text{ where,}$$
$$\text{for } P, P' \in 2^{Q^M}, a \in A^M \setminus I^M,$$
$$P \xrightarrow{a}_{det} P' \iff P' = P \text{ } after\, a.$$

States of $det(M)$, called *meta-states* below, are subsets of Q^{M}, and the initial state q_0^{M} *after* ε is the set of states reachable from q_0^{M} by internal actions. In Sect. 4.3 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, and IF in the case of TGV). The operational semantics of such a language describes all possible behaviours of specifications. This operational semantics is usually implemented in a simulator which allows one to traverse the behaviours of the specification.

We suppose here that the semantics of a specification is given by an IOLTS $S = (Q^{\mathrm{S}}, A^{\mathrm{S}}, \to_{\mathrm{S}}, q_0^{\mathrm{S}})$. The example given in Fig. 2 will be our running example (where τ_i denotes an internal action). It is not a real example, but it will illustrate all particularities of the testing theory and algorithms.

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 model the IUT's behaviours. This is called the *test hypothesis*.

An IUT is modelled by an IOLTS $\mathrm{IUT} = (Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, \to_{\mathrm{IUT}}, q_0^{\mathrm{IUT}})$ 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 *(weakly) input complete*: in each state all inputs are accepted, possibly after internal actions, i.e. $\forall q \in Q^{\mathrm{IUT}}, \forall a \in A_{\mathrm{I}}^{\mathrm{IUT}}, q \overset{a}{\Rightarrow}$. This hypothesis is reasonable when the IUT never refuses an invalid or inopportune input but ignores the request or answers negatively.

3.2 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-hand side of Fig. 3. A *deadlock* state is a state where the system cannot evolve anymore, i.e. $\Gamma(q) = \emptyset$. An *output quiescent* state is a state where the system is waiting only for an input from the environment, i.e. $\Gamma(q) \subseteq A_{\mathrm{I}}^{\mathrm{M}}$. A *livelock* state is a state from which the system diverges by an infinite sequence of internal actions. In the case of the finite-state systems that we consider, a livelock is a loop of internal actions, i.e. $\exists \tau_1, \tau_2, \ldots \tau_n, q \overset{\tau_1.\tau_2 \cdots \tau_n}{\Rightarrow} q$. We denote by $deadlock(M)$ the set of deadlocked states of the IOLTS M, $outputlock(M)$ its set of outputlocks and $livelock(M)$ its set of livelock states. A deadlock is a special case of outputlock; thus $deadlock(M) \subseteq outputlock(M)$. The set of all quiescent states is denoted by

$$quiescent(M) = outputlock(M) \cup livelock(M).$$

As conformance testing is based on the observation of visible behaviours, test synthesis requires a determinization of the specification: two sequences with the 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 possible only if quiescence is computed on the specification.

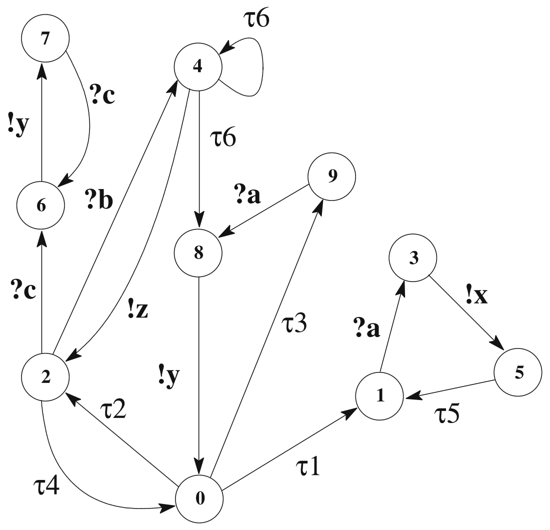

Fig. 2. Specification S

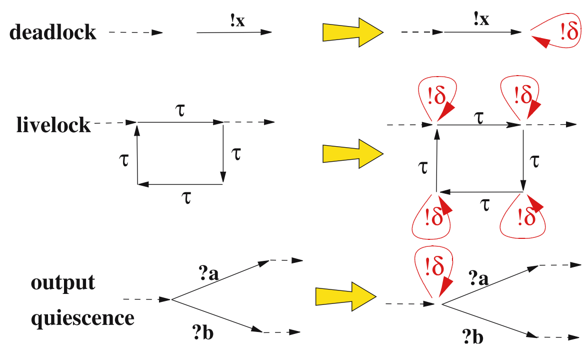

Fig. 3. Quiescence and how to make it explicit

This results in the definition of an IOLTS called a *suspension automaton*, which makes explicit the quiescence by the addition of a new observable action δ, considered as an output. This automaton is described by the following definition, and its construction is sketched in Fig. 3.

Definition 3. *The* suspension automaton *of an IOLTS* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *is an IOLTS* $\Delta(M) = (Q^M, A^{\Delta(M)}, \rightarrow_{\Delta(M)}, q_0^M)$, *where* $A^{\Delta(M)} = A^M \cup \{\delta\}$ *with* $\delta \in A_O^{\Delta(M)}$ *(δ is considered as an output, observable by the environment), and the transition relation* $\rightarrow_{\delta(M)}$ *is obtained from* $\rightarrow_M$ *by adding loops* $q \xrightarrow{\delta} q$ *for each quiescent state q (i.e. livelock or output quiescence and thus deadlock). More formally:*

$$\rightarrow_{\Delta(M)} = \rightarrow_M \cup \{(q, \delta, q) \mid q \in quiescent(M)\}.$$

The traces of $\Delta(S)$ *are called the* suspension traces *of S and are denoted by* $STraces(S)$.

For a specification S, its suspension traces $STraces(S)$ exactly represent all the behaviours of S that can be observed by the environment, i.e. its sequences of inputs, outputs and quiescence. This will thus constitute the basis for test synthesis. The visible behaviour of the IUT is also characterized by its suspension traces $STraces(IUT)$. Conformance testing will thus be based on a comparison of the observed traces $STraces(IUT)$ with expected traces $STraces(S)$, as will be formalized in the next subsection.

Example: Figure 4 represents $\Delta(S)$, the suspension automaton of the specification S of Fig. 2. States 0, 2 and 4 are livelocks as they belong to loops of internal actions, while states 1, 7 and 9 are outputlocks as only inputs are firable in those states.

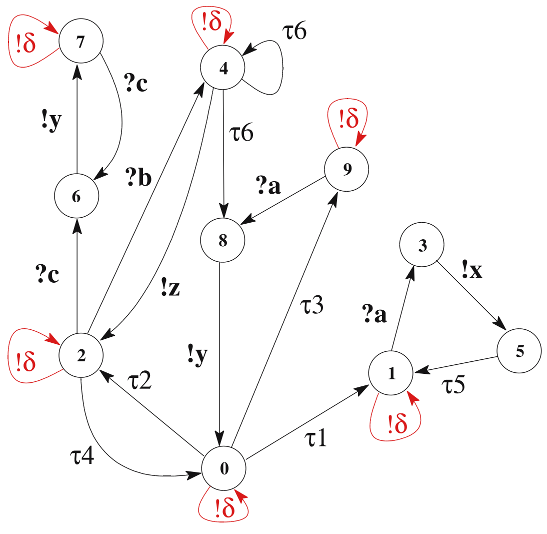

Fig. 4. $\Delta(S)$, the suspension automaton of specification S

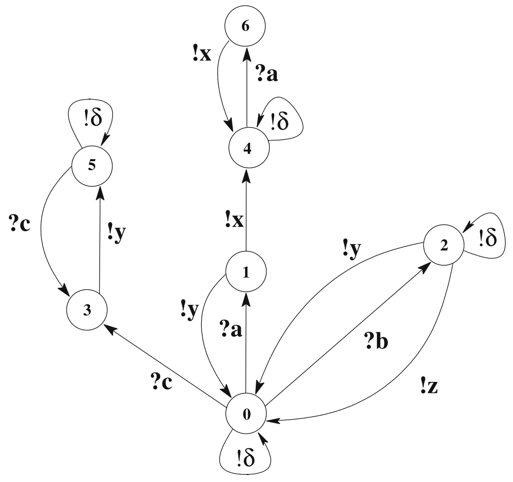

Fig. 5. $det(\Delta(S))$, the visible behaviour of S obtained by determinization of $\Delta(S)$

The suspension traces of S representing the visible behaviour of S are characterized by the sequences of the automaton $det(\Delta(S))$ obtained from $\Delta(S)$ by determinization (Definition 2). For our example, $det(\Delta(S))$ is represented in Fig. 5. Its initial meta-state 0 corresponds to the set of states $0\ after\ \varepsilon = \{0, 1, 2, 9\}$ of $\Delta(S)$. In $\Delta(S)$, $?b$ is firable from state 2, leading to 4. Thus meta-state 0 leads by $?b$ to the meta-state 2 corresponding to the set of states $4\ after\ \varepsilon = \{4, 8\}$. The construction proceeds until no new meta-state is created.

3.3 Conformance relation

A *conformance relation* formalizes the set of IUTs that behave consistently with a specification. Following Tretmans [35], the considered observations during testing are the suspension traces, as they represent the visible behaviour of a system. As the IUT is unknown and conformance, not robustness, is considered the observation is restricted to specified behaviours, and thus to traces of the specification. Intuitively, an implementation IUT conforms to its specification S for **ioco** if after each suspension trace σ of $STraces(S)$ the IUT exhibits only outputs and quiescences that are possible in S. Formally:

Definition 4. *Let S be an IOLTS and IUT be an input complete IOLTS (compatible with S):*

$$\text{IUT}\ \mathbf{ioco}\ S \triangleq \forall \sigma \in STraces(S),$$
$$Out(\Delta(IUT)\ after\ \sigma) \subseteq Out(\Delta(S)\ after\ \sigma).$$

Examples: Figure 6 explains **ioco** for a simple specification and several IUTs. IUT_1 **ioco** S because in each state outputs of IUT_1 are included in outputs of S. **ioco** thus allows us to restrict the IUT on outputs (as in state 1). IUT_1 **ioco** S even if the initial state of IUT_1 allows a new input $?b$, as only the outputs are checked by **ioco**. **ioco** thus allows partial specifications. However,

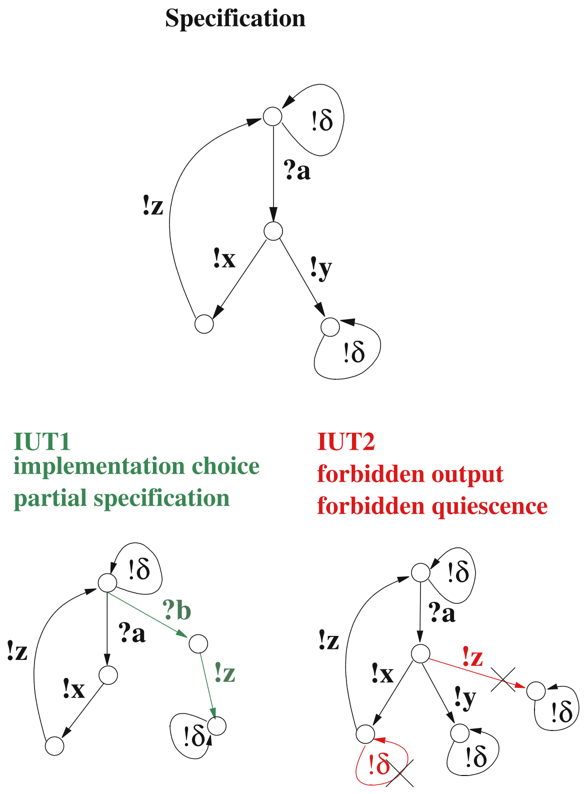

Fig. 6. **ioco** by example: IUT_1 **ioco** S and $\neg(IUT_2$ **ioco** $S)$

$\neg(IUT_2$ **ioco** $S)$, as the output $!z$ after the input $?a$ is not allowed in the specification. The other reason for non-conformance is that the quiescence after $?a!x$ (due to an internal loop for example) is not specified in S.

3.4 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 a non-conformant IUT. For example in Fig. 6, if a tester sends a, it should wait for either x or y, but also for any other output (that will allow the tester to reject IUT_2). In contrast, we assume that the testers do not present choices between outputs as they control them. Furthermore, they have no internal actions. To model a test case, we also use an IOLTS, but extended with verdicts and some additional properties. A test case has a complex behaviour whose structure is a graph with possible loops.

Definition 5. *A* test case *is an IOLTS* $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ *equipped with three sets of trap states* **Pass** $\subseteq Q^{TC}$, **Fail** $\subseteq Q^{TC}$ *and* **Inconc** $\subseteq Q^{TC}$ *characterizing verdicts. Its alphabet is* $A^{TC} = A_I^{TC} \cup A_O^{TC}$, *where* $A_O^{TC} \subseteq A_I^S$ *(TC emits only inputs of S) and* $A_I^{TC} \subseteq A_O^{IUT} \cup \{\delta\}$ *(TC foresees any output or quiescence of IUT). We make several structural assumptions on test cases:*

– *States in* **Fail** *and* **Inconc** *are only directly reachable by inputs:*

$$\forall (q,a,q') \in \rightarrow_{TC} \;\; (q' \in \textbf{Inconc} \cup \textbf{Fail} \Rightarrow a \in A_I^{TC}).$$

– *From each state a verdict must be reachable:*

$$\forall q, \exists \sigma \in A^{TC^*}, \exists q' \in \textbf{Pass} \cup \textbf{Inconc} \cup \textbf{Fail},\; q \xrightarrow{\sigma} q' \,.$$

– *TC is* controllable*: no choice is allowed between two outputs or an input and output:*

$$\forall q \in Q^{TC}, \forall a \in A_O^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \neq a, q \not\xrightarrow{b}_{TC} .$$

– *A test case is* input complete *in all states where an input is possible:*

$$\forall q \in Q^{TC},\; (\exists\, a \in A_I^{TC},\; q \xrightarrow{a}_{TC} \Rightarrow \forall\, b \in A_I^{TC},\; q \xrightarrow{a}_{TC}).$$

A test suite *is a set of test cases.*

Test execution: Test cases are executed against an IUT, and this execution results in verdicts indicating if the IUT should be rejected or not. This execution should be formalized as we need to establish properties such as soundness and exhaustiveness, which relate verdicts of executions to conformance. We assume a synchronous communication between test cases and IUTs. Thus, the execution of a test case against an IUT is modelled by a parallel composition with a synchronization on common visible actions. This is formalized by the three following rules:

$$\frac{p \xrightarrow{a}_P p', q \xrightarrow{a}_Q q'}{(p,q) \xrightarrow{a}_{P||Q} (p',q')},$$

$$\frac{p \xrightarrow{\tau}_P p'}{(p,q) \xrightarrow{\tau}_{P||Q} (p',q)}, \qquad \frac{q \xrightarrow{\tau}_Q q'}{(p,q) \xrightarrow{\tau}_{P||Q} (p,q')}.$$

This model of execution, together with the hypothesis made on the IUT and test cases, ensures that $TC||\Delta(IUT)$ may only block in states where a verdict is returned by TC. Thus verdicts are associated with *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.

Verdicts: A *verdict* associated with the execution of a test case TC on an IUT is completely determined by the state of TC reached by a maximal trace of $TC||\Delta(IUT)$. Depending on this state, it can be *Pass*, *Fail* or *Inconc*:[1]

$$verdict(\sigma) = Fail \triangleq TC \; after \; \sigma \subseteq \textbf{Fail}$$
$$verdict(\sigma) = Pass \triangleq TC \; after \; \sigma \subseteq \textbf{Pass}$$
$$verdict(\sigma) = Inconc \triangleq TC \; after \; \sigma \subseteq \textbf{Inconc}$$

[1] We make a distinction between the verdict, e.g. *Pass*, and the set of states of a test case where a verdict is assigned, e.g. **Pass**.

A possible rejection of an IUT by a test case is defined by:

$$TC\ may\ reject\ IUT \triangleq \exists \sigma \in Traces(TC || \Delta(IUT)),\ verdict(\sigma) = Fail.$$

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.

Test case properties: The execution of test cases on implementations should give a verdict about the conformance of an IUT with respect to a specification. As conformance is defined formally by a conformance relation, we need to relate the verdicts of these executions to the conformance relation. This is done by the following properties of test cases and test suites.

Definition 6. *A test case TC is* sound *for S and* **ioco** *if*

$$\forall IUT, IUT\ \mathbf{ioco}\ S \Rightarrow \neg(TC\ may\ reject\ IUT).$$

A test suite is sound if it consists of sound test cases.

A test suite is exhaustive *for S and* **ioco** *if*

$$\forall IUT, \neg(IUT\ \mathbf{ioco}\ S) \Rightarrow TC\ may\ reject\ IUT.$$

A test suite is complete *if it is both sound and exhaustive.*

The minimal property required for test suites is *soundness*: a test suite should not reject a conformant IUT. This property is important but not sufficient in practice, as test cases accepting all IUTs 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 a non-conformant IUT, it is theoretically possible to produce a test case that may reject it (under some fairness assumption of the IUT).

3.5 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 behaviours to be tested, in general incomplete sequences of actions. In TGV, we model test purposes by automata (formally IOLTS extended with marked states) accepting sequences of actions of the specification. One could restrict test purposes to traces or suspension traces, as advocated in [10]. However, allowing internal actions in test purposes is more powerful. It is very useful when one wants to design test purposes for complex systems when the targeted visible behaviour is difficult to foresee from the behaviours of individual components. This is particularly true when the communication with the system is performed through a context (FIFO channels for example) that provokes a distortion of the IUT's behaviour (Fig. 7). In this case, one would like to test the IUT's behaviour, but its input/output behaviour may not be directly visible by the tester as PCOs are at the boundaries of the system under test. Thus tests cases should be composed of actions which are visible at the PCOs. The specification should describe the whole system under test, including the context. But test purposes can be written according to the input/output behaviour of the specification of the IUT and thus the internal behaviour of the system.

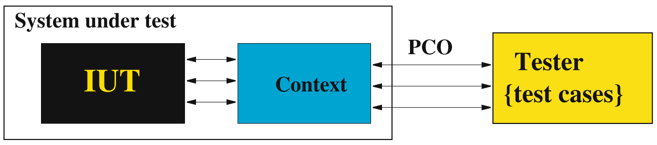

Fig. 7. Testing in context

Another useful feature of test purposes in TGV is the notion of *Accept* and *Refuse* states, allowing an efficient test selection, in particular on-the-fly (Sect. 4.6). *Accept* states are used to select targeted behaviours, while *Refuse* states are used to cut down the exploration of the specification state space when undesired actions are taken. An adequate use of *Refuse* states may dramatically reduce the test generation cost.

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 the 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 loop 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. This is particularly true because in on-the-fly test generation, we want to avoid the construction of the whole state graph of the specification. However, in the above definition, test purposes should be complete, which could seem contradictory. In fact it is not. 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 by convention TGV completes incomplete states by a "*"-loop. This allows the user to describe test purposes with partial sequences of actions that will be automatically completed by TGV. Another abstraction mechanism is provided by the use of regular expressions for the description of sets of labels. This allows us to describe in-

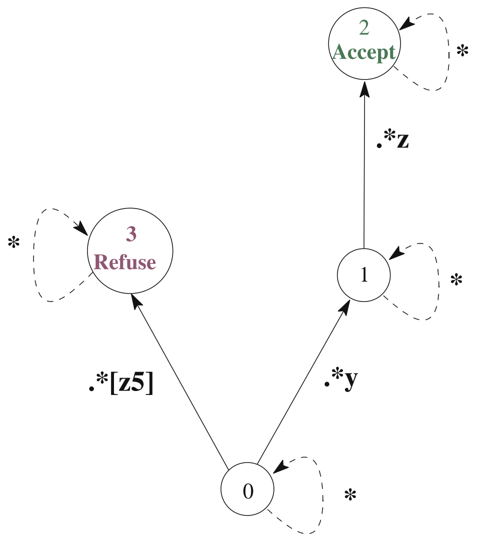

Fig. 8. Test purpose TP for specification S in Fig. 2

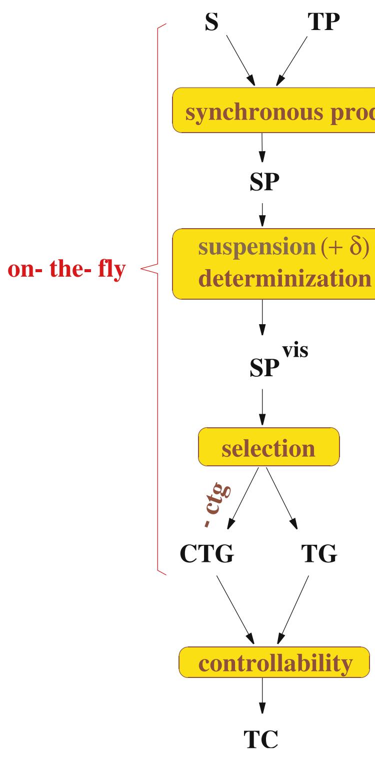

Fig. 9. Overview of test synthesis operations

complete labels of transitions. This is particularly useful as it is sometimes difficult to know the format of transition labels from the specification language. Figure 8 gives an example of a test purpose TP for the specification S. In this example one wants to select sequences of actions in which labels do not end with 5 or z (represented by the regular expression `.*[z5]`) before a y, and when y occurs it is followed by a z. Here "*"-loops are implicit in all states.

4 Principles and algorithms

This section describes the main algorithms of TGV. Let us sketch these algorithms, summarized in Fig. 9. 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 resulting SP we build the visible behaviour (traces and quiescence) in SP^{VIS}. Test selection then builds an IOLTS CTG by extraction of the accepted behaviours 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. This means that the aforementioned IOLTSs do not need to be completely constructed but only partially.

4.1 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* in 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 discussion below, 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 [34] describes an algorithm of linear complexity for the computation of SCCs. In [26], we give an iterative version with "holes" and instantiate these "holes" for several algorithms used in TGV. The algorithm is a depth-first search (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 where an 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.

4.2 Synchronous product

Test synthesis in TGV takes as inputs a specification S and a test purpose TP. The first problem is to identify behaviours of S accepted (on *Accept* states) by TP

or refused (on *Refuse* states) by TP. This is a classical problem of computation of the intersection of languages. Just as in model checking, this is solved by a synchronous product.

Definition 8. *Let* $S = (Q^S, A^S, \rightarrow_S, q_0^S)$ *be an IOLTS and* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ *a test purpose with* $A^{TP} = A^S$ *and equipped with state sets* $Accept^{TP}$ *and* $Refuse^{TP}$*.*
The synchronous product $S \times TP$ *is an IOLTS* $SP = (Q^{SP}, A^{SP}, \rightarrow_{SP}, q_0^{SP})$*, equipped with two disjoint sets of states* $Accept^{SP}$ *and* $Refuse^{SP}$*, and defined as follows:*

- *Its alphabet is* $A^{SP} \triangleq A^S (= A^{TP})$*;*
- *Its state set* Q^{SP} *is the subset of* $Q^S \times Q^{TP}$ *reachable from the initial state* $q_0^{SP} \triangleq (q_0^S, q_0^{TP})$ *by the transition relation* $\rightarrow_{SP}$*;*
- *The transition relation* $\rightarrow_{SP}$ *is defined by:*

$$(q^S, q^{TP}) \xrightarrow{a}_{SP} (q'^S, q'^{TP}) \iff q^S \xrightarrow{a}_S q'^S \wedge q^{TP} \xrightarrow{a}_{TP} q'^{TP};$$

- $Accept^{SP}$ *and* $Refuse^{SP}$ *are defined as follows:*

$$Accept^{SP} \triangleq Q^{SP} \cap (Q^S \times Accept^{TP}),$$
$$Refuse^{SP} \triangleq Q^{SP} \cap (Q^S \times Refuse^{TP}).$$

The effect of the synchronous product is to mark behaviours of S by *Accept* and *Refuse*, and possibly to unfold S. More precisely, accepted behaviours of SP are exactly those behaviours of S which are accepted by TP. As TP is complete, all behaviours of S (including quiescence) are preserved in SP. More precisely, $S \times TP$ is bisimilar to S. SP is built during the following operation but could be built by any traversal.

Figure 10 represents the synchronous product $S \times TP$ of the specification S of Fig. 2 and test purpose TP of Fig. 8. Its suspension automaton $\Delta(S \times TP)$ is obtained by adding the dashed δ loops. The construction has been stopped in *Accept* and *Refuse* states as subsequent behaviours is not explored by TGV as it will be cut by the following operations.

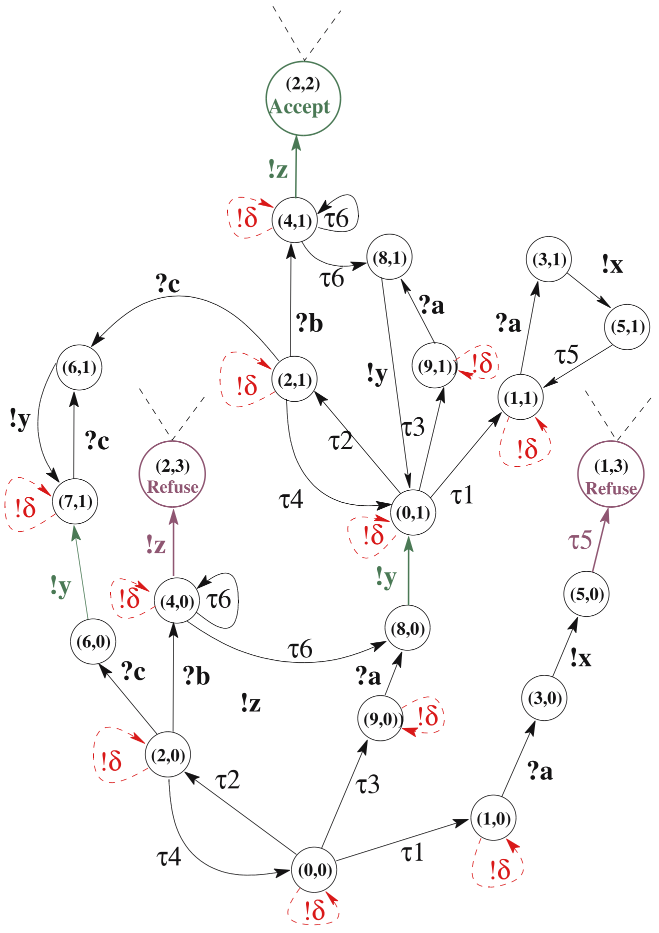

Fig. 10. Synchronous product $SP = S \times TP$ and quiescence $\Delta(SP)$

This choice is justified by the fact that this cuts down the exploration earlier. Figure 11 gives the result of this computation for the examples S of Fig. 2 and TP of Fig. 8. In this example, the exploration has been stopped in *Accept* state 11 and *Refuse* states 4 and 6 as successors of *Accept* states (respectively *Refuse* states) are also *Accept* states (resepctively *Refuse* states).

4.3 Visible behaviours

The next operation consists of 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 $SP^{\mathrm{VIS}} = det(\Delta(SP))$ (Definitions 3 and 2). Note that suspension is applied first because determinization preserves traces, but not quiescence. SP^{VIS} is equipped with *Accept* and *Refuse* states:

$$Refuse^{\mathrm{VIS}} = \{P \in Q^{\mathrm{VIS}} \mid P \cap Refuse^{\mathrm{SP}} \neq \emptyset\},$$
$$Accept^{\mathrm{VIS}} = \{P \in Q^{\mathrm{VIS}} \mid P \cap Accept^{\mathrm{SP}} \neq \emptyset\} \setminus Refuse^{\mathrm{VIS}}.$$

This means that we choose to refuse a trace as soon as it corresponds to at least one refused sequence in SP.

Computation of $det(\Delta(.))$*:* We have already given the definitions of Δ and det, but for the sake of efficiency, quiescence and determinization are computed simultaneously. We will illustrate the computation on $S \times TP$ of Fig. 10 and its result SP^{vis} in Fig. 11.

Theoretically, a δ loop should be added in each quiescent state. For deadlocks (no deadlock in $S \times TP$) and output quiescent states [states (1,0),(1,1), (7,1), (9,0) and (9,1)], we just look at outgoing transitions. For livelocks, which are loops of internal actions [in states (0,0),(0,1), (2,0), (2,1), (4,0) and (4,1)], 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 has the same effect on $\Rightarrow$. We will see how to combine this with determinization.

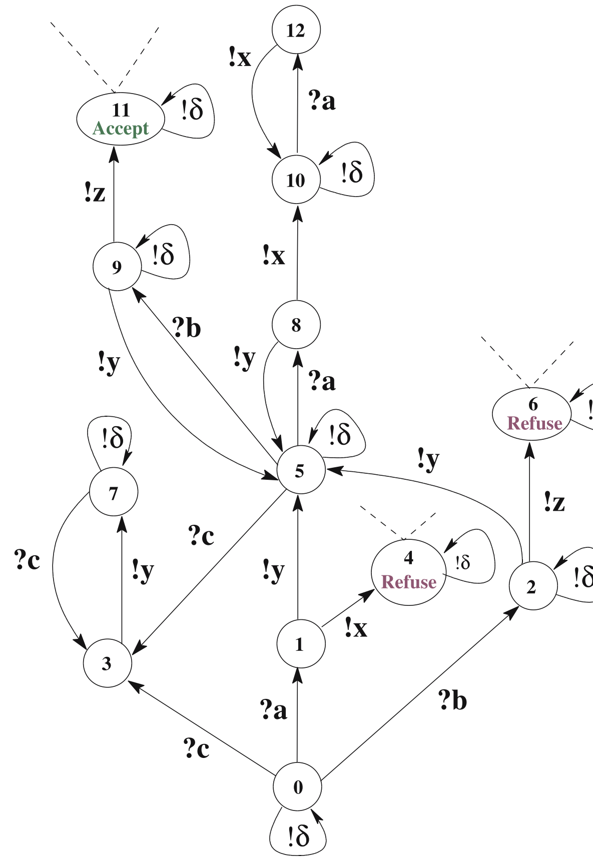

Fig. 11. Visible behaviour of the synchronous product: $SP^{\mathrm{VIS}} = det(\Delta(S \times TP))$

Determinization: Determinization consists of building the deterministic IOLTS $det(\Delta(S))$ starting from its initial meta-state $q_0^{\Delta(S)} = q_0^{\Delta(S)}$ *after* ϵ [in the example, the meta-state 0 of $det(\Delta(S))$ is $\{(0,0),(1,0),(2,0),(9,0)\}$] 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 $P = \{(0,0),(1,0),(2,0),(9,0)\}$ and $?a$, the result is $P' = \{(3,0),(8,0)\}$.
– ε-closure: for a state set P, compute the set P *after* ε of states reachable from P by sequences of internal actions. For $P = \{(3,0),(8,0)\}$, P *after* ε is also $\{(3,0),(8,0)\}$.

In [25] 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 the same τ-SCC, the sets

$$Fire(q) = \bigcup_{a \in A^{\Delta(\mathrm{SP})}} (a, \{q' \mid q \overset{a}{\Rightarrow} q'\})$$

are identical. In fact, $Fire(q)$ denotes visible actions after a τ sequence and resulting states and is thus a reachability property.

For example
$Fire((0,0)) = \{(?a, \{(3,0),(8,0)\}), (?b, \{(4,0)\}),$
$(?c, \{(6,0)\}), (!\delta, \{(0,0),(1,0),(2,0),(9,0)\})\}$
and $Fire((2,0)) = Fire((0,0))$,
while $Fire((1,0)) = \{(?a, \{(3,0)\}), (!\delta, \{(1,0)\})\}$.

A meta-state $[\{(0,0),(1,0),(2,0),(9,0)\}$ for example$]$ is not only a set of states but a reduced graph of τ-SCC $[\{(0,0),(2,0)\}, \{(1,0)\}, \{(9,0)\}]$, and $Fire(root(V_i))$ 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 [e.g. $\{(0,0),(2,0)\}$]. 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

$$Fire(P) = \bigcup_{V_i \in SCC\_init} Fire(root(V_i)),$$

where SCC_{init} is the set of initial SCC of the reduced graph of τ-SCC of P, gives all firable 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 (Sect. 4.6) 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 coding each meta-state P_i by $Fire(P_i)$, which is the only used information in P_i.

4.4 Test selection

SP^{VIS} represents all visible behaviours of S. Among these, some visible behaviours correspond to behaviours accepted (or refused) by the test purpose TP. They are defined by the sets $Accept^{\mathrm{VIS}}$ and $Refuse^{\mathrm{VIS}}$. The next operation consists of extracting a test case by selection of accepted behaviours. This operation does a little more since, to compute a test case (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 the first step, we will not deal with controllability and will describe the computation of an IOLTS *CTG* or *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. Except for **Inconc** and **Fail** states, CTG represents the useful part of SP^{vis}, that is it is composed of states (the set $L2A$) and transitions ($\rightarrow_{L2A}$) playing a role in the acceptance of traces.

Definition 9. *For a specification S and a test purpose TP, the* complete test graph *is an IOLTS $CTG = (Q^{CTG}, A^{CTG}, \longrightarrow_{CTG}, q_0^{CTG})$, with three sets of trap states* **Pass**, **Inconc** *and* **Fail**, *and defined from $SP^{VIS} = det(\Delta(S \times TP))$ as follows:*

– *Its alphabet is $A^{CTG} = A_O^{CTG} \cup A_I^{CTG}$ with $A_O^{CTG} \subseteq A_I^{VIS}$ and $A_I^{CTG} = A_O^{VIS}$ (mirror image).*
– *Its set of states is $Q^{CTG} = L2A \cup$ **Inconc** $\cup$ **Fail**, with*
  – *$L2A = \{q \in Q^{VIS} \mid \exists \sigma \in A^{VIS^*}, v \xrightarrow{\sigma}_{VIS} Accept^{VIS}\}$. $L2A$ stands for* leads to *Accept). It consists of states from which $Accept^{VIS}$ is reachable.*
  – **Inconc** $= \{v \in Q^{VIS} \mid \exists u \in L2A, v \notin L2A, a \in A_O^{VIS}, u \xrightarrow{a}_{VIS} v\}$, *i.e.* **Inconc** *is composed of states not in $L2A$ but which are direct successors of states in $L2A$ by an output in SP^{VIS}.*
  – **Fail** $= \{Fail\}$ *where $Fail \notin Q^{VIS}$ is a new state.*
– *If $q_0^{VIS} \in L2A$, the initial state is $q_0^{CTG} = q_0^{VIS}$ and Q^{CTG} is restricted to states reachable from q_0^{CTG} by $\rightarrow_{CTG}$, otherwise Q^{CTG} is empty.*
– *The transition relation is $\rightarrow_{CTG} = \rightarrow_{L2A} \cup \rightarrow_{\mathbf{Inconc}} \cup \rightarrow_{Fail}$, where*

$$\rightarrow_{L2A} = \rightarrow_{VIS} \cap (L2A \times A^{CTG} \times L2A)$$
$$\rightarrow_{\mathbf{Inconc}} = \rightarrow_{VIS} \cap (L2A \times A_I^{CTG} \times \mathbf{Inconc})$$
$$\rightarrow_{\mathbf{Fail}} = \{(v, a, Fail) \mid v \in L2A \wedge a \in A_I^{CTG} \wedge v \overset{a}{\not\rightarrow}_{VIS}\}.$$

– *Finally,* **Pass** $= Accept^{VIS}$.

Figure 12 illustrates the computation of the complete test graph from SP^{VIS} of Fig. 11 for the examples S and TP. In SP^{VIS}, the SCCs $\{0\}$, $\{1\}$, $\{2\}$, $\{5, 8, 9\}$ and $\{11\}$ all lead to *Accept*, thus their states and transitions are preserved in CTG. $\{3, 7\}$ does not lead to *Accept* and is cut as it is reached by the input $?c$, but outputs $!x$ and $!z$ leading to $\{4\}$, $\{10\}$ and $\{6\}$ are preserved and lead to an Inconclusive verdict.

Algorithm: According to the definition of CTG, the main point is to compute the set $L2A$ and to check if $q_0^{\mathrm{VIS}} \in L2A$. Now, the set $L2A$ consists of co-reachable states of *Accept* (note that all states are reachable from q_0^{VIS} by construction), i.e. states where the CTL [7] property $L2A = EF\,Accept^{\mathrm{VIS}}$ holds. This computation is classical in model-checking and is often performed by a backward traversal from *Accept*. But a forward traversal is possible, using properties of SCC, and is more adapted to on-the-fly computation. As a matter of fact, either all states of an SCC are in $L2A$ or none of them are. The algorithm, called TGVloop, adapts Tarjan's algorithm by the additional synthesis of the attribute $L2A$ and a construction of $\rightarrow_{\mathrm{L2A}}$ during backtracking. This algorithm can be seen as a model-checking algorithm for $L2A$ producing all witnesses of

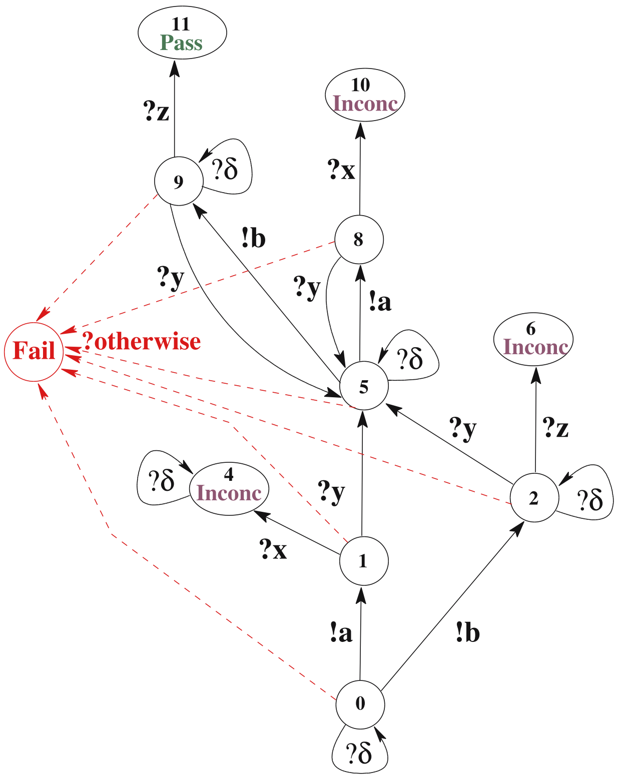

Fig. 12. Complete test graph

$L2A$ starting in the initial state. Moreover, the computation of **Inconc** and $\rightarrow_{\mathbf{Inconc}}$ is done during backtracking of output transitions of SP^{VIS} from states in $L2A$ to states outside $L2A$. The *Fail* state and transitions in $\rightarrow_{\mathbf{Fail}}$ are implicit, $\rightarrow_{fail}$ being defined by complementation of firable transitions. The algorithm has linear complexity in time and space, just like Tarjan's SCC algorithm.

4.5 Pruning 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 (Fig. 13). Solving these conflicts consists of 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. Reachabil-

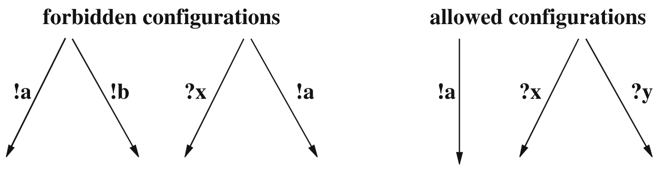

Fig. 13. Controllability conflicts

ity 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**. Figure 14 shows a test case which is one possible result of this conflict resolution for the CTG of Fig. 12. In state 0 of CTG, $!a$ is chosen but $!b$ and $?\delta$ are pruned. In state 5, $!b$ is chosen but $!a$ and $?\delta$ 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 the on-the-fly synthesis (Sect. 4.6). Another solution consists of pruning during backtracking in TGVloop. Suppose that TGVloop pops a state q' of SP^{VIS} from which $Accept^{\mathrm{VIS}}$ is reachable (i.e. $q' \in L2A$), and let q be its immediate predecessor in the DFS stack by action a. If a is an input in SP^{VIS} (an output of the tester), all other transitions leaving q (already explored or not) can be pruned. If a is an output of SP^{VIS} (an input of the tester), all inputs can be pruned. In the first case, already explored transitions do not lead to $Accept^{\mathrm{VIS}}$, otherwise one of the rules would have been applied already and the input a pruned. In the second case, already explored inputs do not lead to $Accept^{\mathrm{VIS}}$, otherwise the first rule would have been applied previously and the input a pruned.

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. In fact, when a state q' is popped from the DFS stack, it is not always known if it is in $L2A$ or not. In the worst case, this is known only when its SCC is computed and the two preceding rules cannot be applied.

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 $!a$.

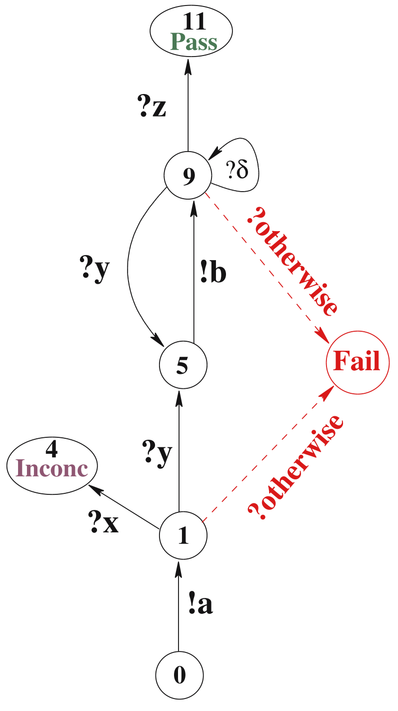

Fig. 14. A possible test case for the example

4.6 On-the-fly test case synthesis

Figure 9 gave an overview of the operations needed for test synthesis. Remember that after the computation of the product $SP = S \times TP$, the suspension automaton $\Delta(SP)$ and the deterministic automaton of it $SP^{\mathrm{VIS}} = det(\Delta(SP))$ are factorized in one operation. *Accept* and *Refuse* sets are propagated by these operations. From SP^{VIS} a complete test graph CTG is computed by selection of traces leading to $Accept^{\mathrm{VIS}}$, mirror image, addition of verdicts. Alternatively, a test graph TG can be built 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 to perform 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* gives the initial state; *firable* gives the set of firable transitions in a state; from a state and a firable transition, *succ* computes the (set of) target state(s). Additionally, a comparison function, and functions computing the membership of *Accept* or *Refuse* are needed.

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 strongly constrains 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 noted, 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.

4.7 Properties of synthesized test suites

Test cases produced by TGV have nice properties relating verdicts computed during test execution and conformance. These properties are stated by the following theorem:

Theorem 1. *For every specification S, all test suites produced by TGV are sound. Moreover, the (infinite) test suite consisting of all test cases that TGV may produce is exhaustive.*

We do not detail the proofs here but just explain the main principles. For soundness, we need to prove that if a test case TC may reject an implementation IUT of a specification S, then $\neg IUT$ **ioco** S. It then suffices to prove that a **Fail** verdict of a test case is only put after a forbidden input (an output of IUT not specified in S) after a suspension trace of S. This is almost clear from the definition of CTG. Conversely, for exhaustiveness, we need to prove that for every non-conformant IUT there is a test purpose TP and a possibility to synthesize a test case TC from S and TP, such that TC may reject IUT. But, if $\neg IUT$ **ioco** S, there is a suspension trace σ of S such that an output of IUT after σ is not possible in S. TP is then constructed from σ and there is a possibility to extract a test case TC from the CTG such that IUT may be rejected by TC.

5 The TGV tool

5.1 TGV architecture

The architecture of TGV follows its functional description (Fig. 15). TGV has several software levels communicating through APIs. Each API is a simulation API of an

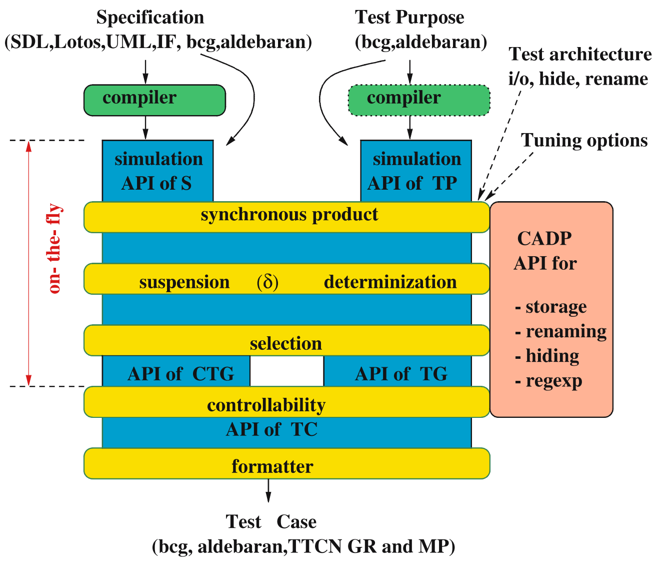

Fig. 15. TGV architecture

IOLTS made of graph traversal functions: the computation of the initial states, the computation of firable transitions of a state, the computation of its successors and the comparison between states. Each level implements one of the algorithms described in Sect. 4, transforming 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 provided by the CADP toolbox [14]. Due to this architecture, TGV guides the simulation API of different specification languages with the same source code, except for the highest API. This ensures the coherency of different variants and facilitates porting to new systems (TGV works on SunOS 5, Linux and WindowsXP). Moreover, some parts can be used alone or by other programs. In particular, we have implemented a module called VTS which verifies soundness and strictness of manual test cases (strictness of a test case states that it may reject all IUTs that are non-conformant on the traces of the test case). This module just replaces TGVloop and uses other levels. It also served for testing TGVloop.

5.2 Supported languages

TGV supports different specification languages 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 ObjectGeode SDL tool from Telelogic [17]. There are two versions of this connection. The academic version uses a CADP-like API and guides 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 the UMLAUT tool [18], a validation framework for UML developed in IRISA. UMLAUT uses class and object diagrams, deployment diagrams, and state diagrams and gives an operational semantics for UML in terms of labeled transition systems by transforming and compiling the UML model.

Another possibility is now offered by the compilation of UML into IF in the context of the Agedis IST project.

IF: IF [5] is an intermediate format developed by Verimag (Grenoble) based on communicating automata extended with data. IF specifications can be produced from

SDL and UML models. TGV also uses the simulation API provided by the IF compiler. Recently, during the IST Agedis project, a new version of this connection was developed. The API offered by IF is now the API of the product between a test directive and the specification. Test directives generalize test purposes in several ways. First, they describe test purposes extended with constraints on data, just as in GOAL observers in ObjectGéode. This allows a more precise test case selection. They also allow one to describe coverage directives similar to those of Gotcha [2]. Coverage directives express coverage of general expressions on specification variables. Covering an expression means finding sequences of transitions covering all reachable values of the expression. These sequences are then transformed into test cases. The main problem is that this may result in the construction of the whole state graph of the specification. This is why coverage directives can also be coupled with test purposes in order to limit the behaviours in which an expression should be covered. Using the general coverage principle, it is also possible to define different state and transition coverage policies. In fact, it suffices to automatically introduce new variables in the IF specification that code state change or transition firing and to cover the values of these variables.

Your favorite specification language: The simulation API required by TGV is documented and quite simple. For a language with an operational semantics described 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.

Output language: TGV may produce test cases in TTCN (Tree and Tabular Combined Notation [19]) or in one of the graph formats (.aut and .bcg) of the CADP toolbox.

5.3 Other TGV characteristics

Several options are provided by TGV in order to tune test generation or to refine the produced test cases. In particular, TGV produces test cases with timer operations. Recall that timers are used to detect quiescence and that quiescence has been taken into account in test generation. Two timers are managed, TAC and TNOAC. TAC is used when no quiescence is expected. Thus, 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. Conversely, TNOAC is used when quiescence is possible. TNOAC is started before entering a state where a quiescence (δ) is allowed. It is cancelled if an input is observed. The observation of δ is replaced by a timeout. This timeout does not produce a *Fail* verdict because the presence of δ proves that quiescence is possible in the specification. This transformation is described by Fig. 16 for the test case in Fig. 14.

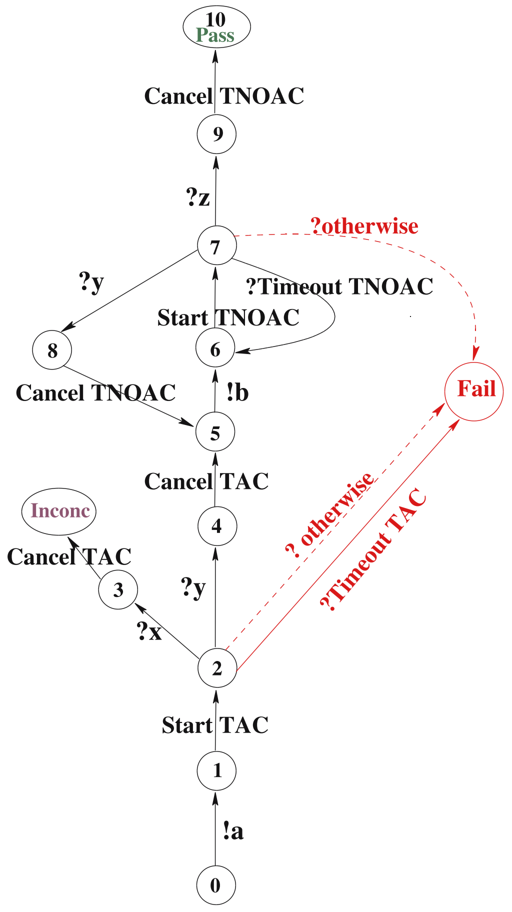

Fig. 16. Test case with timers

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.

The test architecture can also be modified by the use of hiding and renaming rules described by regular expressions. Hiding is used to increase the set of internal actions. Renaming is used to modify the description of visible actions, as they will appear in test cases. These options are useful for studying several test architectures without modifying the original specification. Another option is the description of inputs and outputs. This is particularly useful for specification languages that do not distinguish them, such as Lotos.

6 Case studies

Different versions of TGV have been evaluated on industrial size case studies, in various application domains, and with different specification languages. We just sketch these cases studies, as they have already been published, and summarize the lessons learned from these experiments.

6.1 The DREX protocol

This protocol is a military version of the ISDN D protocol. The study was performed during a project involving several French industrial partners. Several specifications were written in SDL, each describing one service. This case study allowed the validation of TGV principles on a preliminary version of the tool. This version was incomplete and did not work on-the-fly [15]. The same specification served as a case study for two other tools: TVéda from CNET [32] and the TOPIC prototype from Vérilog. The experiment allowed us to compare TGV with these tools as well as with the manual production of test cases [11]. It proved that the TGV approach was feasible on real size case studies. Moreover, it allowed us to detect some errors in manual test cases, in particular some due to asynchronous communication between the tester and the IUT, producing race conditions. These asynchronous behaviours are very difficult to imagine manually and completely justify the use of formal methods.

6.2 A cache coherency protocol

Several experiments [27, 28] consisted of using TGV on Lotos specifications of cache coherency protocols for multiprocessor architectures of Bull (Polykid). The Lotos specification used was 2000 lines long, half of which consisted of abstract data types. The specification consisted of 3 modules with several processes per module. Its state graph could not be built. During the experiment, the first version of TGV working completely on-the-fly was developed. This also allowed us to improve TGV algorithms (generation of acyclic test cases) and to introduce new optional features. In particular, as Lotos does not make a distinction between inputs and outputs, we had to provide this information in an additional file. Renaming and hiding was also intensively used.

Test purposes were written according to a test plan provided by Bull. Seventy-five test cases have been produced from these test purposes. The average size of test cases was 1000 cycles. Produced test cases were then executed by Bull on a simulator of the architecture running a VHDL description of the Polykid architecture. The usual way to perform testing in Bull for hardware architectures was to work off-line, i.e. emulating the system by input data, collecting reactions in output files, analysing the results and emitting a verdict. As test cases produced by TGV are reactive, a testing environment was developed to run these reactive test cases. The test campaign uncovered five bugs mainly due to address collisions.

6.3 The SSCOP protocol

The SSCOP protocol (service specific connection oriented protocol) is a quite complex protocol of the ATM stack, standardized by ITU. This protocol is supposed to transfer data between two high-bandwidth network entities. A specification was coded in SDL by FTR&D from the ATM Forum specification. It consisted of one single process describing several services. Its size was approximately 2000 lines of textual SDL (approximately 80 pages of graphical SDL). We have used this protocol and its SDL specification in several experiments with the aim of putting into relief the particularities of TGV [4]. The version of TGV that was used was connected to the ObjectGéode SDL simulator (Telelogic). So we used ObjetGéode features to tune the experiments. In particular, we imposed restrictions on the environment behaviour with the use of feeds. We also used GOAL observers to specify the global service automaton of the ATM standard. This was useful for detecting errors in the specification and for ensuring that sequences traversed by TGV during test case generation did not violate the service. We also used static analysis to safely reduce the specification state graph. Fifty complex test purposes were designed, covering all services of the protocol, but of course not all behaviours. We made some variations on the number of PCOs and the communication mode (synchronous or asynchronous) between tester and IUT. Asynchronous communication was specified by the introduction of a process between the system and the environment. The results gained during these experiments showed that on-the-fly test generation was efficient on specifications with large state spaces. This resulted in the transfer of TGV into ObjectGéode.

We also used the same SDL specification to check for correctness of a part of the TTCN test suite produced by the ATM Forum. For this, we used our tool VTS, which is built from parts of TGV. VTS takes as input a specification and a test case and checks for soundness and strictness of the test case [24]. This allowed us to detect some errors with respect to soundness in the ATM test suite. Most of them were due to asynchronism.

6.4 The conference protocol

TGV has also been used on a conference protocol [12]. This protocol is a toy example designed by colleagues at Twente University to compare test generation tools. Several specifications have been written in different languages, including SDL and Lotos. Also, 28 mutants of a correct implementation were written in order to check if tools were able to detect non-conformant mutants. An experiment with TorX had already been conducted on a Lotos specification, and TorX was able to detect all non-conformant mutants. A new experiment was then conducted with TGV on the same Lotos specification, during a visit of colleagues from Twente, in order to compare TGV with TorX. The challenge was to detect all non-conformant mutants by running generated test cases. Of course, the code of mutants was not available to us. The main problem encountered with TGV was to imagine adequate test purposes. Of course, this involves a good knowledge of the protocol as one has to imagine abstract scenarios where at least one implementation may fail. We first used informal requirements provided with the protocol to write test purposes and were able to detect most non-conformant IUTs by generated test cases. The last non-conformant mutants were more difficult to find as faults occurred after long sequences involving loops in protocol entities. But finally, after a careful study of the protocol, new test purposes were written, and all non-conformant IUTs were detected.

Another experiment with the SDL version of the protocol was conducted later using the version of TGV in TestComposer.

6.5 Air traffic controller

A UML model of an air traffic control (ATC) system was used as an example of the UMLAUT/TGV connection. This model consists of a class diagram consisting of four classes and three actors, one state diagram per class or actor, and object diagrams specifying the initial state. The environment behaviour is defined by actors. One describes a human controller, the second describes the radar and the third describes a controller of another ATC. The four classes describe the flight and flight plan, the position of flights and the flight plan manager. The semantics of a UML model in UMLAUT is defined by a labeled transition system obtained by transformations of the UML model. Simple test purposes have been automatically generated from sequence diagrams. From these, TGV produced interesting test cases. The case study was done to demonstrate that test generation using TGV was possible for UML models.

6.6 Transit Computerization Project

In the framework of the IST European project Agedis, TGV has been used on an IF specification of the ECN component of the Transit Computerization Project (TCP). The aim of TCP is to develop a set of applications to be used for electronic exchange of information regarding goods in transit between EU countries. The ECN is mainly in charge of ensuring the communication and translation of business information flows between domains. From an informal UML model of the system provided by IntraSoft, an SDL specification was written by Verimag and then automatically translated into IF. The specification consists of ten processes running concurrently and communicating asynchronously. Two experiments were conducted. The first one with TGV in TestComposer, directly with the SDL specification, used test purposes generated with branch coverage and test purposes generated with interactive simulation. This experiment showed that branch coverage was clearly not sufficient to cover most interesting behaviours. Thus additional test purposes were designed by simulation and from requirements. The second experiment was done with TGV connected with the IF simulator, using the IF specification and a few significant test purposes. The number of processes (ten) and their concurrency pushed TGV to its limits. In particular, we noticed that there were a lot of concurrencies between internal actions. But these concurrencies could be avoided, as test generation is concerned with visible behaviour. This gave us some ideas about possible improvements using partial-order methods (Sect. 8). Finally, we generated a state graph of the specification with additional constraints. The size of the graph was of the order of 500 000 states and 900 000 transitions.

6.7 Lessons learned from case studies

We have sketched some case studies in which members of our team participated. TGV has been used by us or some of our partners in other case studies in telecommunications but also for smart card applications.

First, one notices that we made some realistic case studies in very different domains. TGV was first designed for telecommunication protocols but showed that it could also be applied to hardware as well as to middleware. This proves that the TGV approach is very general. The reason is that the testing theory and algorithms are general enough for all these application domains.

We also used different specification languages. This clearly shows the independence of TGV with respect to specification languages. This is not surprising as all these languages are given a semantics in terms of labelled transition systems. An additional interpretation of actions in terms of internal, input or output actions is sometimes necessary but is often clear.

On-the-fly test generation has proved useful in most cases. In fact, sometimes state graphs of specifications were infinite, but in most cases they were very large, due to data and/or asynchronism between processes, and thus impossible to build completely. Nevertheless, we were able to generate test cases with TGV. Of course, if state graphs can be completely constructed, on-the-fly test generation is not necessary. But TGV can still be used on these explicit state graphs.

All experiments were useful for imagining improvements of TGV. Test generation algorithms have been improved over the first version. Starting from algorithms generating acyclic test cases, TGV now generates test cases with loops and takes into account coverage directives. We also improved the tuning of TGV by the addition of options in test generation algorithms.

The main difficulty in most case studies was to design test purposes. In some cases, the task was easier as we could base test purpose design on requirements. But this was not always the case. In a first approach one writes very abstract test purposes. But it is often necessary to refine these test purposes for several reasons. First, when state graphs are large, abstract test purposes do not guide the generation sufficiently. Thus TGV may suffer from state space explosion. Second, even if a test case is produced, one realizes that a shorter one could be generated. This implies restricting test purposes by the addition of refuse states or limiting the depth of test cases.

Nevertheless, we know that test generation from test purposes is not always the best approach for some users. Some prefer a more automatic solution based on coverage criteria. But coverage is limited as it often misses interesting behaviours. Thus additional test cases based on well-targeted test purposes are often necessary. However, we are concerned with coverage. This is why we recently tried to improve TGV with coverage facilities, allowing us to mix coverage directives with test purposes.

7 Comparison with other techniques and tools

7.1 FSM-based test generation

FSM test generation tools make strong assumptions about specifications and implementations. This is the price to pay for exhaustiveness, as this restricts the set of possible implementations to a finite set. This means that if a fault is present, it can be detected after a bounded number of steps. This corresponds to a regularity hypothesis in the framework of [3]. Conversely, the assumptions made by TGV are very weak. The only significant one is that implementations are input complete. In practical terms, exhaustiveness cannot be assured because the set of possible implementations is infinite. This means that a fault can occur after a trace of arbitrary length and thus cannot be detected by test cases of bounded length. However, all faults are detectable as proven by Theorem 1.

Moreover, FSM-based test generation algorithms are complex and are thus limited to small specifications. Usually, when large specifications are considered, a rough abstraction is made, or the state graphs are built up to a limited depth. Thus exhaustiveness is only partial.

7.2 Test generation based on model checking

TGV can be compared with test synthesis techniques and tools based on model checking (e.g. [13]). The common idea of most of these techniques is to use a standard model checker to produce counter-examples. Given a test purpose specified by a reachability property P of a temporal logic (e.g. LTL or CTL), a model checker (e.g. SPIN, SMV) is used to produce a witness of P on the specification S. To do this, one checks the negation $\neg P$ of the property against the specification S. The property $\neg P$ is a safety property that can be violated by a finite trace. Most model checkers can produce counter-examples for this kind of property. Thus, if S violates $\neg P$ (thus P is satisfied by S), the model checker produces a counter-example for $\neg P$ and thus a witness for P. This witness is then abstracted from internal actions and interpreted as a test case. TGV goes beyond this idea. First, it is based on a clear testing theory. Second, it does not use a model-checking tool but adapts model-checking algorithms to test synthesis. This allows us to take into account non-deterministic and non-controllable specifications, which is not the case for other tools.

7.3 TorX

The most comparable tool for TGV is TorX [9] 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 of the IUT for the proposed stimuli. As mentioned in Sect. 5.3, both tools were applied to the same case study and, despite their differences, gave similar results in terms of fault detection power. In some sense, TGV algorithms are more powerful than TorX ones for test selection. They both base test generation on a traversal of suspension traces of the specification. But while TorX works forward and randomly, TGV works both forward and backward guided by a test purpose. Nevertheless, the approaches of TGV and TorX are complementary. TorX is very efficient for intensive testing, when the goal is to detect faults by a random exploration of behaviours. TGV is more efficient when precise faults are targeted by a test purpose.

8 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 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 a higher abstraction level and because TGV ensures soundness of synthesized test cases. But an effort must be made for the description of test purposes, and this requires some expertise. TestComposer provides a partial answer by the synthesis of test purposes according to a coverage criterion adapted from branch coverage but limited to observable behaviours. But the branch coverage criterion 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. In the context of the Agedis project, we improved TGV with test directives that include both test purposes and coverage criteria (e.g. state and transition coverage).

Improvements of algorithms are also to be investigated. An interesting direction is to use partial-order techniques as in model checking [30]. 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, and thus on synthesized test cases. Applying these techniques for visible actions is more difficult as concurrent behaviour must be synthesized in test cases. Other improvements concern compositionality. We will investigate how to compute test cases incrementally in the case of compositional specifications. In the same line of thought, in the context of Agedis we also investigated how to compute several test cases in one run from a composition of test purposes or coverage criteria.

Another important problem is that of distributed testing. In the general case the system is distributed, and test cases should be distributed and should communicate asynchronously. Concurrent-TTCN has such specification power. A first approach we adopted [23] was to synthesize a sequential test case and to distribute it according to localities of actions. Global choices were solved by a distributed consensus service. The main drawback is the loss 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 [21, 22] 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. Also, specifications with symbolic variables are beyond the scope of TGV. A solution is to use symbolic techniques [33]. State 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 symbolic constants, variables and communication parameters, and may perform assignments. From a specification specified as an IOSTS and test purpose (with *Accept* and *Refuse* states) also specified by an IOSTS, a test case is first 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. Unfortunately, this problem is undecidable, and thus approximate methods must be used. In our tool STG [6], we use two means. We use the Omega constraint solver to prune some locally unsatisfiable transitions. Moreover, a deeper analysis using abstract interpretation (by our NBAC tool) computes an over-approximation of reachable and co-reachable states, which prunes more unsatisfiable transitions. Even if some unsatisfiable transitions remain, after fixing the values of symbolic constants, executable test cases can be produced and executed on implementations. Omega is again used during execution to find outputs satisfying the guards.

Acknowledgements. The TGV tool is the result of common work over the course of several years. We wish to thank all participants in its design and development at Irisa and Verimag: Erwan Demairy, Jean-Claude Fernandez, Alain Kerbrat, Pierre Morel, Laurence Nedelka, Joseph Sifakis, Séverine Simon, Valery Tschaen, 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.

We also wish to thank the reviewers of the paper for their helpful comments that allowed us to improve the paper.

References

1. Abramsky S (1987) Observational equivalence as a testing equivalence. Theor Comput Sci 53(3):225–241
2. Benjamin M, Geist D, Hartman A, Mas G, Smeets R, Wolfsthal Y (1999) A feasibility study in formal coverage driven test generation. In: 36th Design Automation conference, DAC99, June 1999
3. Bernot G, Gaudel MC, Marre B (1991) Software testing based on formal specification: a theory and a tool. Softw Eng J 6:387–405
4. Bozga M, Fernandez J-C, Ghirvu L, Jard C, Jéron T, Kerbrat A, Morel P, Mounier L (2000) Verification and test generation for the SSCOP protocol. J Sci Comput Programm 36(1):27–52
5. Bozga M, Graf S, Mounier L (2002) IF-2.0: A validation environment for component-based real-time systems. Lecture notes in computer science, vol 2404. Springer, Berlin Heidelberg New York, pp 343–348
6. Clarke D, Jéron T, Rusu V, Zinovieva E (2002) STG: a symbolic test generation tool. In: International conference on tools and algorithms for construction and analysis of systems (TACAS2002), Grenoble, France, April 2002. Lecture notes in computer science, vol 2280. Springer, Berlin Heidelberg New York
7. Clarke E, Emerson EA (1981) Synthesis of synchronisation skeletons for branching time temporal logic. In: Workshop in logic of programs, (Yorktown Heights, NY). Lecture notes in computer science, vol 131. Springer, Berlin Heidelberg New York
8. De Nicola R, Henessy M (1984) Testing equivalences for processes. Theor Comput Sci 34:83–133
9. De Vries RG, Tretmans J (2000) On-the-fly conformance testing using Spin. Int J Softw Tools Technol Transf 2(4):382–393
10. De Vries RG, Tretmans J (2001) Torwards formal test purposes. In: Brinskma E, Tretmans J (eds) Workshop FATES'01: Formal Approaches of Testing of Software. BRICS Notes Series NS-01-4
11. Doldi L, Encontre V, Fernandez J-C, Jéron T, Le Bricquir S, Texier N, Phalippou M (1996) Assessment of automatic generation methods of conformance test suites in an industrial context. In: Baumgarten B, Burkhardt A, Giessler H-J (eds) IFIP TC6 9th international workshop on testing of communicating systems, September 1996. Chapman & Hall, London
12. du Bousquet L, Ramangalahy S, Simon SVC, Belinfante A, De Vries RG (2000) Formal test automation: the conference protocol with TGV/TorX. In: Ural H, Probert R, v. Bochmann G (eds) IFIP 13th international conference on testing of communicating systems (TestCom 2000). Kluwer, Dordrecht

13. Engels A, Feijs L, Mauw S (1997) Test generation for intelligent networks using model-checking. In: Third Workshop TACAS, Enschede, The Netherlands, Lecture notes in computer science, vol 1217. Springer, Berlin Heidelberg New York
14. Fernandez J-C, Garavel H, Kerbrat A, Mateescu R, Mounier L, Sighireanu M (1996) CADP: A protocol validation and verification toolbox. In: Alur R, Henzinger TA (eds) Proc. CAV'96 New Brunswick, NJ, August 1996. Lecture notes in computer science, vol 1102. Springer, Berlin Heidelberg New York
15. Fernandez J-C, Jard C, Jéron T, Viho G (1997) An experiment in automatic generation of conformance test suites for protocols with verification technology. Sci Comput Programm 29:123–146. Egalement disponible en rapport de recherche Irisa n^o 1035 et Inria no 2923
16. Gaudel M-C, James PR (1999) Testing algebraic data types and processes : a unifying theory. Formal Aspects Comput 10(5–6):436–451
17. Groz R, Jéron T, Kerbrat A (1999) Automated test generation from SDL specifications. In: Dssouli R, von Bochmann G, Lahav Y (eds) SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec, June 1999, pp 135–152. Elsevier, Amsterdam
18. Ho W-M, Jézéquel J-M, Le Guennec A, Pennaneac'h F (1999) UMLAUT: an extendible UML transformation framework. In: Proc. Automated Software Engineering (ASE'99), Florida, October 1999
19. ISO (1992) Information Technology – Open Systems Interconnection Conformance Testing Methodology and Framework. International Standard ISO/IEC 9646-1/2/3. Part 1: General Concept – Part 2: Abstract Test Suite Specification – Part 3: The Tree and Tabular Combined Notation (TTCN)
20. ITU (1996) ISO/IEC JTC1/SC21 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
21. Jard C (2002) Principles of test synthesis using true-concurrency models. In: König H, Schiferdecker I (eds) Proc. Testcom'2002, Berlin, March 2002
22. Jard C (2003) Synthesis of distributed testers from true-concurrency models of reactive systems. Int J Inf Softw Technol 45:791–888
23. Jard C, Jéron T, Kahlouche H, Viho C (1998) Towards automatic distribution of testers for distributed conformance testing. In: FORTE/PSTV'98, Paris, France, November 1998. Chapman & Hall, London
24. Jard C, Jéron T, Morel P (2000) 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, August 2000. Kluwer, Dordrecht
25. Jéron T, Morel P (1997) 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, September
26. Jéron T, Morel P (1999) Test generation derived from model-checking. In: Halbwachs N, Peled D (eds) CAV'99, Trento, Italy, July 1999. Lecture notes in computer science, vol 1633. Springer, Berlin Heidelberg New York, pp 108–122
27. Kahlouche H, Viho C, Zendri M (1998) An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In: Petrenko A, Yevtushenko N (eds) IFIP TC6 11th international workshop on testing of communicating systems, September. Chapman & Hall, London
28. Kahlouche H, Viho C, Zendri M (1999) Hardware testing using a communication protocol conformance testing tool. In: Cleaveland WR (ed) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), March 1999. Lecture notes in computer science, vol 1579. Springer, Berlin Heidelberg New York, pp 315–329
29. Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – a survey. Proc IEEE 84(8):1090–1123
30. Peled D (1994) Combining partial order reductions with on-the-fly model-checking. In: Dill DL (ed) CAV Workshop. Lecture notes in computer science, vol 818. Springer, Berlin Heidelberg New York
31. Petrenko A (2000) Fault model-driven test derivation from finite state models: annotated bibliography. In: Cassez F, Jard C, Rozoy B, Ryan M (eds) MOVEP'2k MOdelling and VErification of Parallel processes, Nantes, France. Lecture notes in computer science, vol 2067. Springer, Berlin Heidelberg New York, pp 196–205
32. Phalippou M (1994) Test sequence generation using Estelle or SDL structure information. In: FORTE'94, Berne, October
33. Rusu V, du Bousquet L, Jéron T (2000) An approach to symbolic test generation. In: Integrated Formal Methods (IFM'00), Dagstuhl, Germany, November 2000. Lecture notes in computer science, vol 1945. Springer, Berlin Heidelberg New York, pp 338–357
34. Tarjan R (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1:146–160
35. Tretmans J (1996) Test generation with inputs, outputs and repetitive quiescence. Softw Concepts Tools 17:103–120