Property Oriented Test Case Generation

Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon

Verimag, - Centre Equation - 2 avenue de Vignate - F38610 Gieres, France
{Jean-Claude.Fernandez,Laurent.Mounier,Cyril.Pachon}@imag.fr,
http://www-verimag.imag.fr/

Abstract. In this paper we propose an approach to automatically produce test cases allowing to check the satisfiability of a linear property on a given implementation. Linear properties can be expressed by formulas of temporal logic. An observer is built from each formula. An observer is a finite automaton on infinite sequences. Of course, testing the satisfiability of an infinite sequence is not possible. Thus, we introduce the notion of bounded properties. Test cases are generated from a (possibly partial) specification of the IUT and the property to validate is expressed by a parameterised automaton on infinite words. This approach is formally defined, and a practical test generation algorithm is sketched.

1 Introduction

Testing is certainly one of the most popular software validation techniques and it is a crucial activity in many domains such as embedded systems, critical systems, information systems, telecommunication, etc. Consequently, a lot of work was carried out during the last decade to both formalise the testing activities and to develop tools allowing to automate the production and execution of test suites.

The particular problem of testing if an implementation is "correct" with respect to its specification is referred to as *conformance testing*. This problem was mainly investigated inside the telecommunication area (as described in the ISO standard 9646 [7]), and a formal approach was outlined in [19, 11, 3]. These works gave birth to several (academic and commercial) tools[1, 12, 17, 6] able to automatically generate test cases from a system specification. For instance, in [5], a technique is proposed to derive test cases from a (formal) specification and a test purpose. This technique is based on a partial exploration of a kind of product between the specification and the test purpose. An associated tool, called TGV, was developed by Irisa Rennes and Verimag Grenoble.

We first explicit a bit more the concepts of *black box testing* and *conformance testing*.

Black box testing. We consider here "black box" testing, meaning that the behaviour of the *IUT* (*Implementation Under Test*) is only visible by an external tester, through a restricted test interface (called PCO, for *Points of Control and Observation*). There exists two kinds of interactions between the tester and the IUT: outputs of the tester are stimuli sent in order to control the IUT, whereas

inputs of the tester are observations of the IUT's outputs. These sets of interactions are described by a *test architecture*. In black box testing the internal state of the IUT is not observable by the tester. Consequently:

- the tester cannot observe the internal non-determinism of the IUT ;
- the tester should remain deterministic since it cannot backtrack the IUT to a given internal state.

A possible model to describe these sequences of interactions is Input-Output Labelled Transition System (IOLTS, see definition below).

Conformance testing. Conformance testing is based on the following concepts:

- *IUT*: Even if the internal code of the IUT is not visible from the outside, its behaviour can be characterised by its interactions with its environment. This external behaviour can be modeled with an IOLTS. We suppose in the following that this IOLTS is *input complete*, that is, in each state the IUT cannot refuse any input proposed by the environment.
- *Test architecture*: Test architecture defines the set of interactions between the IUT and an environment, distinguishing between *controllable* and *observable* events. ISO9646 standard proposes four test methods : local test method, distributed test method, coordinated test method, and remote test method. All these methods are based on the black box testing principles and describe possible environment of the IUT. Of course, test architecture is a parameter of a test generation technique. In this paper, we consider a local test architecture.
- *Specification*: The specification represents the expected behaviour of the IUT, to be used as a reference by the tester. This expected behaviour can be also formally modelled by an IOLTS. Note that the specification not necessarily describes only the *visible* behaviour of the IUT, but it may also contain some of the internal actions performed by the implementation.
- *Conformance relation*: Defining whether an IUT is correct or not with respect to a specification is achieved in this context by introducing a formal relation between IOLTS. Several relations have been proposed so far, such as **ioco** [19]. Other relations have also been proposed on other models (such **conf** [3]).
- *Test case*: Roughly speaking a test case is a set of interactions (input and output) sequences a tester can perform on an IUT interface. When executed, each interaction sequence delivers a *verdict* indicating whether the IUT was found conform or not on this particular execution (with respect to a given conformance relation).
- *Test purpose*: The test purpose represents a particular functionality (or sets of abstract scenarios) the user wants to test. It can also be modeled by an IOLTS, and may be used to automate the test case generation.

Although this conformance testing framework is now well established and happens to be be very useful in the telecommunication area, its use in other application domains suffers, in our opinion, from two important limitations:

– first, it requires a rather *exhaustive* formal specification, since conformance is defined with respect to this specification, and any IUT exhibiting unexpected behaviours (from this specification point of view) would be rejected;
– second, the conformance relation is not very *flexible*: it is not always easy to understand what it does exactly preserve, and, more important, it is not possible to adapt it to the particular functionality one wants to test.

We propose in this work to extend this framework (and particularly what was done inside the TGV tool) to the generation of *property oriented* test cases. The general idea is to allow automatic test generation from a *partial specification* (not necessarily expressing the overall expected behaviour of the system), and with respect to a particular property (test case execution should indicate whether the IUT satisfy or not this property). This approach is outlined below.

Property testing. The properties we consider are *linear* properties: each property defines a language (i.e., a set of sequences), and an IUT satisfies a given property if and only if all its execution sequences belong to its associated language. In this context it is a common practice to distinguish between *safety* properties, that can be checked by considering only finite execution sequences of the IUT, and *liveness* properties that need to consider also the infinite ones. Several characterisations of such properties have been proposed in the verification community, based on various specification formalisms : automata on infinite words (recognising ω-regular languages), linear-time temporal logics (or μ-calculus), boolean equation systems, etc. Automata on infinite words, like Büchi automata [4], are very interesting from an algorithmic point of view, and they are used in several decision procedures [20] implemented in model checkers. It can be shown in particular that any ω-regular language can be characterised by a Büchi automaton, or, equivalently, by a *deterministic* Rabin automaton, see for example [9, 16]. Since the use of a deterministic automaton is an important issue in the test generation technique we propose in this paper, we will consider in the following that the property to be checked is expressed by a deterministic Rabin automaton. Of course, testing the satisfiability of a liveness property is not possible: it would require an infinite execution time. However, automata on infinite words can be parameterised to specify so-called *bounded liveness* properties: the automaton recognises a set of infinite execution sequences and some external parameters simply limit the "length" of the sequences to consider (this length being expressed for instance in number of interactions, or as an overall execution time).

More precisely, the test generation technique we propose can be sketched as follows:

– A (possibly partial) specification $\mathcal{S}$ is used as a "guideline" for the test case synthesis, and it is therefore supposed to be "closed enough" to the actual behaviour of the IUT. Note however that we do not require at this level any particular conformance relation between $\mathcal{S}$ and the IUT.

– A safety or bounded liveness property $\mathcal{P}$ is given through an observer $\mathcal{O}$bs recognising sequences of $\neg\mathcal{P}$. This observer is a parameterised automaton on infinite words.
– Test cases are automatically generated by traversing the specification in order to find the "most promising" execution sequences able to show the non satisfiability of $\mathcal{P}$ by the IUT. These execution sequences are the sequences recognised by $\mathcal{O}$bs that are the "closest" to the ones provided by the specification.

Related Work. Producing test cases from a formal specification to check the satisfiability of a given property is a rather natural idea, and consequently numerous works have been already carried out in this area, leading to various kinds of tools. They mostly differ in the nature of the specification and property they consider, and they are often based on probabilities to select the test sequences (such in [15, 10, 8, 13]). However, an original aspect of our approach is the use of parameterized automata on infinite words to specify properties and to instanciate them only at test time. In addition, test cases we produce are IOLTS (not only sequence sets) that can be executed against non deterministic IUTs.

2 Models

This section formalises the different elements involved in the test case generation framework we propose.

2.1 Input-Outputs labelled transition systems

The basic models we consider are based on Input-Output Labelled Transition Systems (IOLTS), namely Labelled Transition Systems in which input and output actions are differentiated (due to of the asymmetrical nature of the testing activity). We consider a finite alphabet of actions A, partitioned into two sets: *input actions* A_I and *output actions* A_O. A (finite) IOLTS is a quadruplet $\mathrm{M}=(Q^{\mathrm{M}}, A^{\mathrm{M}}, T^{\mathrm{M}}, q^{\mathrm{M}}_{\mathrm{init}})$ where Q^{M} is the finite set of states, $q^{\mathrm{M}}_{\mathrm{init}}$ is the initial state, $A^{\mathrm{M}} \subseteq A$ is a finite alphabet of actions, and $T^{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A^{\mathrm{M}} \cup \{\tau\} \times Q^{\mathrm{M}}$ is the transition relation. Internal actions are denoted by the special label $\tau \notin A$. τ is assumed to be unobservable for the system's environment whereas actions of A^{M} are visible actions representing the interactions either between the system and its environment, or between its internal components.

Notations. We denote by $\mathbb{N}$ the set of non negative integers. For each set X, X^* (resp. $X^\omega = [X \to \mathbb{N}]$) denotes the set of finite (resp. infinite) sequences on X. Let $\sigma \in X^*$; σ_i or $\sigma(i)$ denotes the i^{th} element of σ. We adopt the following notations and conventions: Let $\sigma \in A^*$, $\alpha \in A$, $p, q \in Q^{\mathrm{M}}$. We write $p \xrightarrow{\alpha}_{\mathrm{M}} q$ iff $(p, \alpha, q) \in T^{\mathrm{M}}$ and $p \xrightarrow{\sigma}_{\mathrm{M}} q$ iff $\exists \sigma_1, \sigma_2 \cdots \sigma_n \in A$, $p_0, \cdots, p_n \in Q^{\mathrm{M}}$ such that $\sigma = \sigma_1.\sigma_2 \ldots \sigma_n$ and $p_0 = p$, $p_i \xrightarrow{\sigma_{i+1}}_{\mathrm{M}} p_{i+1}$ for $i < n$, $p_n = q$. In this case, σ is called a *trace* or *execution sequence*, and $p_0 \cdots p_n$ a *run* over σ. An infinite run

of M over an infinite execution sequence σ is an infinite sequence ρ of Q^{M} such that 1. $\rho(0) = q^{\mathrm{M}}_{\mathrm{init}}$ and 2. $\rho(i) \stackrel{\sigma(i)}{\rightarrow}_{\mathrm{M}} \rho(i+1)$). $\mathbf{inf}(\rho)$ denotes the set of symbols from Q^{M} occurring infinitely often in ρ: $\mathbf{inf}(\rho)=\{q \mid \forall n.\ \exists i.\ i \geq n. \wedge\ \rho(i) = q\}$.

Let V a subset of the alphabet A. We define a *projection operator* $\downarrow_V$: $A^* \rightarrow V^*$ in the following manner: $\epsilon \downarrow_V = \epsilon$, $(a.\sigma) \downarrow_V = \sigma \downarrow_V$ if $a \notin V$, and $(a.\sigma) \downarrow_V = a.(\sigma \downarrow_V)$ if $a \in V$. This operator can be extended to a language L (and we note $L \downarrow V$) by applying it to each sequence of L. The language recognised by M is $\mathcal{L}(M) = \{w \mid \exists q$ such that $q^{\mathrm{M}}_{\mathrm{init}} \stackrel{w}{\rightarrow} q\}$.

Let $\mathrm{M}=(Q^{\mathrm{M}}, A^{\mathrm{M}}, T^{\mathrm{M}}, q^{\mathrm{M}}_{\mathrm{init}})$ an IOLTS, we recall the *completeness, determinism* and *quiescence* notions.

Completeness. M is *complete with respect to a set of actions* $X \subseteq A$ if and only if for each state q^{M} of Q^{M} and for each action x of X, there is at least one outgoing transition of T^{M} from q^{M} labelled by $x \in X$:

$\forall p^{\mathrm{M}} \in Q^{\mathrm{M}} \cdot \forall x \in X \cdot \exists q^{\mathrm{M}} \in Q^{\mathrm{M}}$ such that $p^{\mathrm{M}} \stackrel{x}{\rightarrow}_{\mathrm{M}} q^{\mathrm{M}}$.

Determinism. M is said *deterministic with respect to a set of actions* X if and only if it is a deterministic IOLTS containing only actions labelled by elements of X:

$\forall p^{\mathrm{M}} \in Q^{\mathrm{M}} \cdot \forall x \in X \cdot p^{\mathrm{M}} \stackrel{x}{\rightarrow}_{\mathrm{M}} q^{\mathrm{M}} \wedge p^{\mathrm{M}} \stackrel{x}{\rightarrow}_{\mathrm{M}} q'^{\mathrm{M}} \Rightarrow q^{\mathrm{M}} = q'^{\mathrm{M}}$.

We introduce a *determinisation operator*

det (M,X)$=(Q^{\mathrm{det\ (M,X)}}, A^{\mathrm{det\ (M,X)}}, T^{\mathrm{det\ (M,X)}}, q^{\mathrm{det\ (M,X)}}_{\mathrm{init}})$ to compute a deterministic IOLTS with respect to X associated to M. This IOLTS is defined as follows: $Q^{\mathrm{det(M,X)}} \subseteq 2^{Q^M}$, $A^{\mathrm{det(M,X)}} = X$, $q^{\mathrm{det(M,X)}}_{init} = \{q \in Q^M \mid q^M_{init} \stackrel{\omega}{\rightarrow}_{\mathrm{M}} q \wedge \omega \in (A \setminus X)^*\}$ and $T^{\mathrm{det(M,X)}} = \{(S_p, a, S_q) \mid \exists p \in S_p.\ \exists q \in S_q.\ p \stackrel{\omega.a}{\rightarrow}_{\mathrm{M}} q$ with $a \in X \wedge \omega \in (A \setminus X)^*\}$. Note that, $\mathcal{L}(M) \downarrow X = \mathcal{L}(\mathrm{det}\,(M, X))$.

Quiescence. A test should be able to observe IUT quiescence [19]. Several kinds of quiescence may happen: a state p is said *quiescent* in M either if it has no outgoing transition (deadlock), or if it belongs to a cycle of internal transitions (livelock):

$$quiescent\,(p) \equiv (\nexists (a, q).\ p \stackrel{a}{\rightarrow}_{\mathrm{M}} q) \vee p \stackrel{\tau^+}{\rightarrow}_{\mathrm{M}} p$$

Quiescence can be modelled at the IOLTS level by introducing an extra transition to each quiescent state labelled by a special symbol δ. δ is considered as an output (observable by the environment). In practice, the quiescence is observed by means of timers: a timeout occurs if and only if the implementation is locked inside a quiescent state. Formally, we handle quiescence by associating to LTS M its so-called "suspension automaton" $\delta\,(M) = (Q^M, A^M \cup \{\delta\}, T^{\delta(M)}, q^M_0)$ where $T^{\delta(M)} = T^M \cup \{(p, \delta, p) \mid p \in Q^M \wedge$ quiescent $(p)\}$

2.2 Specification and implementation

The system specification is in general expressed using a dedicated language or notation (SDL, Lotos, UML, etc). The operational semantics of this language can be described in terms of IOLTS. Thus, we note the specification S=$(Q^{\mathrm{S}}, A^{\mathrm{S}}, T^{\mathrm{S}}, q^{\mathrm{S}}_{\mathrm{init}})$, with $A^{\mathrm{S}} = A^{\mathrm{S}}_I \cup A^{\mathrm{S}}_O$.

The Implementation Under Test (IUT) is assumed to be a "black box" those behaviour is known by the environment only through a restricted interface (a set of inputs and outputs). From a theoretical point of view, it is convenient to consider the IUT behaviour as an IOLTS IUT=$(Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, T^{\mathrm{IUT}}, q^{\mathrm{IUT}}_{\mathrm{init}})$, where $A^{\mathrm{IUT}} = A^{\mathrm{IUT}}_I \cup A^{\mathrm{IUT}}_O$ is the IUT interface. We assume in addition that this IUT is complete with respect to to A_I (it never refuses an unexpected input), and that the specification S is *a partial IOLTS* of the IUT:

$$A^{\mathrm{S}} \subseteq A^{\mathrm{IUT}} \text{ and } \mathcal{L}(S) \subseteq \mathcal{L}(\mathrm{IUT}) \downarrow (A^{\mathrm{S}}).$$

Intuitively, a specification is partial if each trace of the specification may be executed by the IUT (but the IUT may contain unspecified behaviours).

2.3 Property and satisfiability relation

The objective of this work is to generate test cases allowing to check the satisfiability of some classes of *properties* on a given IUT. In particular we restrict ourselves to *linear properties*, those associated *models* are sets of IOLTS execution sequences. Two kinds of linear properties can be considered: *safety* properties, characterised by *finite* execution sequences, and *liveness* properties, characterised by *infinite* ones. Thus, an IUT will satisfy a given linear property $\mathcal{P}$ if and only if all of its execution sequences belong to the model of $\mathcal{P}$.

From the test point of view, only the (non-)existence of a finite execution sequence can be checked on a given IUT (since the test execution time has to remain bounded). This restricts in practice the test activity to the validation of safety properties. Nevertheless, an interesting sub-class of safety properties are the so-called *bounded liveness*. Such properties allow for instance to a express that the IUT will exhibit a particular behaviour within a given amount of time, or before a given number of iterations has been reached. From a practical point of view, it is very useful to express such properties as liveness (i.e., in terms of infinite execution sequences, telling that the expected behaviour will *eventually* happen), and then to bound their execution only at test time. The main advantage is that the "bounds" are not part of the test generation process, and they can be chosen depending on the concrete test conditions. Therefore, we propose in this section to specify the properties of interest using a general model, allowing to express both finite and infinite execution sequences. This model is then "parameterised" to handle bounded liveness properties.

Automata on infinite words. Several acceptance conditions (Büchi, Muller, Streett, Rabin, etc) have been proposed to extend finite-state IOLTS to recognise infinite sequences. We recall the definition of Büchi and Rabin automata and illustrate on an example the difference between them.

Definition 1. *A Büchi automaton R_b is a structure (B,$\mathcal{G}^B$) where $B = (Q^B, A^B, T^B, q^B_{init})$ is an IOLTS and $\mathcal{G}^B$ is a subset of Q^B. The automaton R_b* accepts *an infinite execution σ of $A^{B\omega}$ if there is an infinite run ρ of B over σ such that* $\mathbf{inf}(\rho) \cap \mathcal{G}^B \neq \emptyset$.

Definition 2. *A Rabin automaton R_a is a structure (R,$\mathcal{T}^R$) where $R = (Q^R, A^R, T^R, q^R_{init})$ is an IOLTS and $\mathcal{T}^R = \langle (L^R_1, U^R_1), (L^R_2, U^R_2), \ldots, (L^R_k, U^R_k) \rangle$ is a pairs table with $L^R_i, U^R_i \subseteq Q^R$ for $i \in \{1, 2, \ldots, k\}$. The automaton R_a* accepts *an infinite execution σ of $A^{R\omega}$ if there is an infinite run ρ of R over σ such that for some $i \in \{1, 2, \ldots, k\}$, $\mathbf{inf}(\rho) \cap L^R_i \neq \emptyset$ and $\mathbf{inf}(\rho) \cap U^R_i = \emptyset$.*

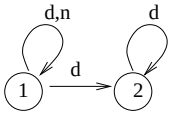

Fig. 1. Non deterministic Büchi automaton recognising $(d+n)^*d^\omega$

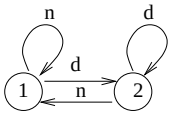

Fig. 2. Deterministic Rabin automaton recognising $(d+n)^*d^\omega$

Example. As an example, consider the following property "*The system always comes back to its nominal mode (action n) after entering a degraded one (action d)*". This property can be expressed by the following (ω-regular) language: $L = (d^*n)^\omega$. The negation of this property is expressed by $\overline{L} = (d+n)^*d^\omega$ which is not recognisable by a deterministic Büchi automaton. The non deterministic Büchi automaton recognising $\overline{L}$ is given by the figure 1, with $\mathcal{G}^B = \{2\}$ and the initial state is 1.

Consider now the deterministic automaton of figure 2 as a Büchi automaton, with $\mathcal{G}^B = \{2\}$ and the initial state is 1. This automaton accepts all sequences containing infinitely often many occurrences of n or many occurrences of d, which are not in $\overline{L}$.

Now, if the automaton of figure 2 is considered as a Rabin automaton with the pair table $\{\{2\}, \{1\}\}$, then this automaton recognises exactly $\overline{L}$ (it accepts an infinite word iff it has infinitely many occurrences of d). Thus, we consider in

this paper *deterministic Rabin automata* [14] since they recognise all classes of ω-regular language.

As another example, the figure 3 shows a Rabin automata with pair (L,U) equals to ($\{3\}$, $\emptyset$) recognising execution sequences in which a `req` action is at some point followed by an `error` action. The δ-loop on state 3 indicates that a finite execution sequence terminating after an `error` action is recognised by this automaton. This artefact allows to deal both with finite and infinite execution sequences.

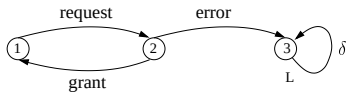

Fig. 3. Example of a safety property expressed by a Rabin automaton

Rabin automata are a natural model to express liveness properties. However, to correctly handle bounded liveness as well, we need to "parameterise" these automata in order to limit the size of the infinite execution sequences they recognise. The (simple) solution we propose consists in associating a counter to each state belonging to an (L_i, U_i) pair. An execution sequence σ is now recognised if and only if it visits "sufficiently often" an L_i-state, and "not too often" an U_i-state, according to the counters associated to these sets (those actual value will be instantiated at test time).

Definition 3. *A Parameterised Rabin automaton is a tuple* $PR_a = (R, \mathcal{T}^R, \mathcal{C}^R)$ *where* $(R, \mathcal{T}^R)$ *is a Rabin automaton and* $C = \{(cl_1, cu_1), \ldots, (cl_k, cu_k)\}$ *with* $cl_i, cu_i \in \mathbb{N}$. *An execution sequence* σ *is accepted by* PR_a *if and only if: there is an finite run of* PR_a ρ *on* σ *such that for some* $i \in \{1, 2, \ldots, k\}$

$|\{j \mid \rho(j) \in L_i^R\}| \geq cl_i$ *and* $|\{j \mid \rho(j) \in U_i^R\}| \leq cu_i$

Thus, the language accepted by PR_a *is* $\mathcal{L}(PR_a)$, *the set of sequences accepted by* PR_a.

Observer and satisfiability relation. Test case generation with respect to a linear property $\mathcal{P}$ is facilitated by considering an *observer automaton* recognising exactly the execution sequences of $\neg\mathcal{P}$. Since we want to deal with safety and bounded liveness properties we choose here to model these observers as deterministic Parameterised Rabin automaton $\mathcal{O}\text{bs} = (\mathcal{O}, \mathcal{T}^o, \mathcal{C}^o)$. We are now able to formally define the *satisfiability relation* relation we consider between an IUT and a linear property.

Definition 4. *Let IUT be an IOLTS,* $\mathcal{P}$ *a property, and* $\mathcal{O}bs = (\mathcal{O}, \mathcal{T}^o, \mathcal{C}^o)$ *the observer recognising the sequences of* $\neg\mathcal{P}$, *where* $\mathcal{O} = (Q^o, A^o, T^o, q^o_{init})$. *Then, IUT* satisfies $\mathcal{P}$ *iff* $(\mathcal{L}(IUT) \downarrow A^o) \cap \mathcal{L}(\mathcal{O}) = \emptyset$. *That is, none of the observable execution sequences of the IUT are recognised by the observer.*

2.4 Test architecture and test case

Test Architecture. At the abstract level we consider, a test architecture is simply a pair ($\mathcal{A}_c$, $\mathcal{A}_u$) of actions sets, each of them being a subset of $\mathcal{A}$: the set of *controllable* actions $\mathcal{A}_c$, initiated by the tester, and the set of *observable* actions $\mathcal{A}_u$, observed by the tester. A test architecture will be said *compliant* with an observer $\mathcal{O}$bs if it satisfies the following constraints : $A_I^{\mathcal{O}} \subseteq A_c$ and $A_O^{\mathcal{O}} \subseteq A_u$. In other words the tester needs at least being able to control (resp. observe) all inputs (resp. outputs) appearing in the observer.

Test Cases. For a given observer $\mathcal{O}$bs, a test architecture (A_c, A_u) compliant with $\mathcal{O}$, an (abstract) test case is a Parameterised Rabin automaton (TC,$\mathcal{T}^{\mathrm{TC}}, \mathcal{C}^{\mathrm{TC}}$) with TC=($Q^{\mathrm{TC}}, A^{\mathrm{TC}}, T^{\mathrm{TC}}, q^{\mathrm{TC}}_{\mathrm{init}}$) and satisfying the following requirements:

1. $A^{\mathrm{TC}} = A_I^{\mathrm{TC}} \cup A_O^{\mathrm{TC}}$ with $A_O^{\mathrm{TC}} = A_c$ and $A_I^{\mathrm{TC}} \subseteq A_u$. Note that A_c (resp. A_u) is the set of controllable (resp. observable) actions.
2. TC is deterministic wrt A^{TC}, controllable (for each state of Q^{TC} there is at most one outgoing transition labelled by an action of A_c), and input-complete (for each state of Q^{TC}, for each element a of A_u, there exists exactly one outgoing transition labelled by a).
3. The pair table $\mathcal{T}^{\mathrm{TC}} = \langle (L_1^{\mathrm{TC}}, U_1^{\mathrm{TC}}), \ldots, (L_k^{\mathrm{TC}}, U_k^{\mathrm{TC}}) \rangle$ is defined with respect to the observer $\mathcal{O}$bs:
   $q^{\mathrm{TC}} \in L_i^{\mathrm{TC}}$(resp. U_i^{TC}) iff $\exists \sigma \in A^{\mathrm{TC}*}, q^{\mathrm{O}} \in L_i^{\mathrm{O}}$(resp. U_i^{O}) such that $q^{\mathrm{O}}_{\mathrm{init}} \stackrel{\sigma}{\rightarrow} q^{\mathrm{O}}$ and $q^{\mathrm{TC}}_{\mathrm{init}} \stackrel{\sigma\downarrow A^{\mathrm{O}}}{\rightarrow} q^{\mathrm{TC}}$

The last condition expresses that there is an execution sequence of the test case starting from the initial state of the test case and leading to a state of L_i^{TC} (resp. U_i^{TC}) if there is a corresponding execution sequence of the observer starting from the initial state of the observer and leading to a state of L_i^{O} (resp. U_i^{O}).

2.5 Test cases execution and verdicts

Let IUT=($Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, T^{\mathrm{IUT}}, q^{\mathrm{IUT}}_{\mathrm{init}}$) an implementation, (TC,$\mathcal{T}^{\mathrm{TC}}, \mathcal{C}^{\mathrm{TC}}$) a test case with TC=($Q^{\mathrm{TC}}, A^{\mathrm{TC}}, T^{\mathrm{TC}}, q^{\mathrm{TC}}_{\mathrm{init}}$), and ($A_c$, A_u) a test architecture. The test execution of TC on IUT can be modelled by a parallel composition between IUT and TC with synchronisations on action sets A_c and A_u. More formally this test execution can be described by an IOLTS $\mathcal{E}$=($Q^{\varepsilon}, A^{\varepsilon}, T^{\varepsilon}, q^{\varepsilon}_{\mathrm{init}}$), where $A^{\varepsilon} = A^{\mathrm{TC}}$, and sets Q^{ε} and T^{TC} are defined as follows:

- Q^{ε} is a set of *configurations*. A *configuration* is a triplet $(p^{\mathrm{TC}}, p^{\mathrm{IUT}}, \lambda)$ where $p^{\mathrm{TC}} \in Q^{\mathrm{TC}}, p^{\mathrm{IUT}} \in Q^{\mathrm{IUT}}$ and λ is a partial function from Q^{TC} to $\mathbb{N}$, which counts the number of times an execution sequence visits a state belonging to L_i^{TC} or U_i^{TC}.
- T^{ε} is the set of transitions $(p^{\mathrm{TC}}, p^{\mathrm{IUT}}, \lambda) \stackrel{a}{\rightarrow}_{\varepsilon} (q^{\mathrm{TC}}, q^{\mathrm{IUT}}, \lambda')$ such that
  - $p^{\mathrm{TC}} \stackrel{a}{\rightarrow}_{\mathrm{TC}} q^{\mathrm{TC}}$, $p^{\mathrm{IUT}} \stackrel{a}{\rightarrow}_{\mathrm{IUT}} q^{\mathrm{IUT}}$ and

- $\lambda'(q^{\mathrm{TC}}) = \begin{cases} \lambda(q^{\mathrm{TC}}) & \text{if } q^{\mathrm{TC}} \notin \bigcup_{i\in\{1,\cdots k\}} (L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}) \\ \lambda(q^{\mathrm{TC}}) + 1 & \text{if } q^{\mathrm{TC}} \in \bigcup_{i\in\{1,\cdots k\}} (L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}) \end{cases}$

The initial configuration $q_{\mathrm{init}}^{\mathrm{TC}}$ is $(q_{\mathrm{init}}^{\mathrm{TC}}, q_{\mathrm{init}}^{\mathrm{IUT}}, \lambda_{\mathrm{init}})$, where for all q, $\lambda_{\mathrm{init}}(q) = 0$.

T^{ε} describes the interactions between the IUT and the test case. Each counter associated with a state of $L_i^{\mathrm{TC}} \cup U_i^{\mathrm{TC}}$ is incremented when an execution sequence visits this state.

Verdicts. Test execution is supposed to deliver some *verdicts* to indicate whether the IUT was found correct or not. These verdicts can be formalised as a function on runs of $\mathcal{E}$ to the set $\{\mathbf{Pass}, \mathbf{Fail}\}$. More precisely:

Fail The execution of a run ρ of $\mathcal{E}$ on σ gives the verdict **Fail** if and only if there is an $i \in \{1, 2, \ldots, k\}$ and a $l \in \mathbb{N}$ such that
1. $\rho(l) = (p_l^{\mathrm{TC}}, p_l^{\mathrm{IUT}}, \lambda_l)$, $p_l^{\mathrm{TC}} \in L_i^{\mathrm{TC}}$ and $\lambda_l(p_l^{\mathrm{TC}}) \geq cl_i$, and
2. for each $m \in [0 \cdots l]$ $\rho(m) = (q_m^{\mathrm{TC}}, q_m^{\mathrm{IUT}}, \lambda_m)$ satisfies $\lambda_m(q_m^{\mathrm{TC}}) \leq cu_i$. In this case, the property is not satisfied.

Pass Similarly, the execution of a run ρ give the verdict **Pass** iff $\forall i \in \{1, 2, \ldots, k\}$ and $\forall l \in \mathbb{N}$
1. $\rho(l) = (p_l^{\mathrm{TC}}, p_l^{\mathrm{IUT}}, \lambda_l)$ and $p_l^{\mathrm{TC}} \in L_i^{\mathrm{TC}}$ implies $\lambda_l(p_l^{\mathrm{TC}}) < cl_i$ or
2. there is a $m \in [0 \cdots l]$ $\rho(m) = (q_m^{\mathrm{TC}}, q_m^{\mathrm{IUT}}, \lambda_m)$ and $\lambda_m(q_m^{\mathrm{TC}}) > cu_i$.

In practice the test case execution can be performed as follows:

- At each step of the execution the controllability condition may give a choice between a controllable and an observable action. In this situation the tester can first wait for the observable action to occur (using a local timer), and then choose to execute the controllable one.
- Formal parameters $\mathcal{C}^{\mathrm{TC}}$ are instantiated according to the actual test environment. Counters are then associated to each sets U_i^{TC} and L_i^{TC}. These counters, initialised to 0, are incremented (inside the test case) whenever a corresponding state is reached during test execution. Thus, a **Fail** verdict is issued as soon as an incorrect execution sequence is reached (according to definition above), and a **Pass** verdict is issued either if the current execution sequence visits "too many often" a state of U_i^{TC} $(\lambda_m(q_m^{\mathrm{TC}}) > cu_i)$, or if a global timer, started at the beginning of test execution, expires. This last case occurs when an execution sequence enter a loop without state belonging to L_i^{TC} or U_i^{TC}.

3 Test Generation

We propose in this section an algorithm to automate the generation of "property oriented" test cases. This algorithm takes as input a (partial) specification S_0 of

a given implementation IUT, an observer (a deterministic parameterised Rabin automaton) $\mathcal{O}\text{bs} = (O, \mathcal{T}^{\mathcal{O}}, \mathcal{C}^{\mathcal{O}})$ characterising the negation of a linear property $\mathcal{P}$), and a test architecture $TA = (A_c, A_u)$. Test cases produced by this algorithm are *sound* in the sense that, when executed against the IUT, a **Fail** verdict is produced *only if* this IUT does not satisfy property $\mathcal{P}$.

The test generation algorithm we propose is based on two steps: generation of a so-called *test graph* (TG, for short) and *test cases selection* from this TG. We first describe these two steps at an abstract level, and then we discuss some implementations issues.

3.1 Test graph

The purpose of the test graph is to gather a set of execution sequences, computed from the specification S_0 and the observer $\mathcal{O}\text{bs}$, compliant with the test architecture TA (i.e., executable by an external tester), and able to witness the non satisfiability of $\mathcal{P}$ for a given IUT. Each controllable sub-graph of this TG could then be turned into an executable test case for property $\mathcal{P}$ (as defined in the previous section).

However, even for a simple property and with a restricted test architecture, it appears that the number of sequences matching this definition is quite large: in fact it could be *any* sequence over $A_c \cup A_u$ recognised by $\mathcal{O}$. Considering such a "complete" test graph would be of limited practical interest in this context: most of these sequences are likely to be very "far" from the actual IUT behaviour, and executing them would not provide very useful information. Consequently, the probability to extract a "relevant" controllable test case from this large set would be rather low. Therefore we need some heuristic to restrict this test graph to the most promising execution sequences.

The heuristic we propose here to compute the test graph is to exploit at best the information provided by the specification, proceeding as follows:

1. First, we transform the initial specification S_0 by computing its deterministic suspension automaton S with respect to the test architecture TA: $S = \delta\left(\det\left(S_0, A_c \cup A_u\right)\right)$. This operation preserves the observable/controllable language of the specification: $\mathcal{L}(S) = \mathcal{L}(S_0) \downarrow (A_c \cup A_u)$.
2. Then, we select the longest sequences of $\mathcal{L}(\mathcal{S})$ matching with a prefix of $\mathcal{L}(\mathcal{O})$. Such sequences are the most promising candidates to witness the non-satisfiability of $\mathcal{P}$) since they belong both to the specification (and then are supposed to be executable on the IUT), and to a prefix of $\mathcal{L}(\mathcal{O})$.
3. Finally, these sequences are then extended to cover complete sequences of $\mathcal{L}(\mathcal{O})$. Note that if the specification already contains a complete sequence of $\mathcal{L}(\mathcal{O})$ (and not only one of its proper prefix) this means that the specification itself does not satisfy $\mathcal{P}$.

From a more formal point of view the test graph we compute is a parameterised Rabin automaton $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$: the IOLTS TG gathers the set of execution sequence described above, and the pair table $\mathcal{T}^{TG}$ and counter sets

$\mathcal{C}^{TG}$ are inherited from $\mathcal{T}^{\mathcal{O}}$ and $\mathcal{C}^{\mathcal{O}}$. This is described in definition 5 below, proceeding in two steps:

1. Computation of an asymmetric product $\otimes$ between S and $\mathcal{O}$. The purpose of this product is to mark each state p_S of S with a corresponding state $p_{\mathcal{O}}$ of $\mathcal{O}$, such that p_S and $p_{\mathcal{O}}$ are reachable from the initial states by "matching" execution sequences (rules R1 and R2).
2. Selection of the longest execution sequences of $S \otimes \mathcal{O}$ matching with a prefix of $\mathcal{L}(\mathcal{O})$, and extension of these sequences to obtain a complete sequence of $\mathcal{L}(\mathcal{O})$. This is performed by rule R4: a transition $(p_S, p_{\mathcal{O}}) \xrightarrow{a} (p_S, q_{\mathcal{O}})$ is added to the transition relation T^{TG} iff such a transition exists in $\mathcal{O}$ but not in $S \otimes \mathcal{O}$.

Definition 5. *Let $TA = (A_c, A_u)$ a test architecture, S_0 a specification and $S{=}(Q^s, A^s, T^s, q^s_{init})$ its deterministic suspension automaton with respect to TA: $S = \delta(det(S_0, A_c \cup A_u))$. Let $(\mathcal{O}, \mathcal{T}^{\mathcal{O}}, \mathcal{C}^{\mathcal{O}})$ be an observer with $\mathcal{O}{=}(Q^o, A^o, T^o, q^o_{init})$ and $\mathcal{T}^{\mathcal{O}} = \langle (L^o_1, U^o_1), (L^o_2, U^o_2), \ldots, (L^o_k, U^o_k)\rangle$ such that TA is* compliant *with $\mathcal{O}$. We define the Parameterised Rabin automaton $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ where $TG{=}(Q^{TG}, A^{TG}, T^{TG}, q^{TG}_{init})$, such that $Q^{TG} \subseteq Q^S \times Q^{\mathcal{O}}$, $A^{TG} \subseteq A^S$, $q^{TG}_0 = (q^S_0, q^{\mathcal{O}}_0)$, and Q^{TG}, T^{TG} are obtained as follows:*

1. *Let $Q^{\otimes}$ and $T^{\otimes}$ be the smallest sets satisfying rules R0, R1 and R2 below:*

$$q^{TG}_0 \in Q_{\otimes}\ [\mathrm{R0}]$$

$$\frac{(p_S, p_{\mathcal{O}}) \in Q^{\otimes},\ p_S \xrightarrow{a}_{T^S} q_S,\ p_{\mathcal{O}} \xrightarrow{a}_{T^{\mathcal{O}}} q_{\mathcal{O}}}{(q_S, q_{\mathcal{O}}) \in Q^{\otimes},\ (p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T^{\otimes}} (q_S, q_{\mathcal{O}})}[R1]$$

$$\frac{(p_S, p_{\mathcal{O}}) \in Q^{\otimes},\ p_S \xrightarrow{a}_{T^S} q_S,\ \neg p_{\mathcal{O}} \xrightarrow{a}_{T^{\mathcal{O}}}}{(q_S, p_{\mathcal{O}}) \in Q^{\otimes},\ (p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T^{\otimes}} (q_S, p_{\mathcal{O}})}[R2]$$

2. *Then, Q^{TG} and T^{TG} are the smallest sets satisfying rules R3 and R4 below:*

$$Q^{\otimes} \subseteq Q^{TG},\ T^{\otimes} \subseteq T^{TG}\ [\mathrm{R3}]$$

$$\left.\frac{\begin{array}{c}(p_S, p_{\mathcal{O}}) \in Q^{TG},\\ p_{\mathcal{O}} \xrightarrow{a}_{T^{\mathcal{O}}} q_{\mathcal{O}},\ \sigma \in (A^S \setminus A^{\mathcal{O}})^*\\ \nexists q_S.\ ((p_S, p_{\mathcal{O}}) \overset{\sigma.a}{\Longrightarrow}_{T^{\otimes}} (q_S, q_{\mathcal{O}}))\end{array}}{(p_S, q_{\mathcal{O}}) \in Q^{TG},\ (p_S, p_{\mathcal{O}}), \xrightarrow{a}_{T^{TG}}, (p_S, q_{\mathcal{O}})}\right\}[\mathrm{R4}]$$

3. *The pair table $\mathcal{T}^{TG}$ is equal to $\langle (L^{TG}_1, U^{TG}_1), (L^{TG}_2, U^{TG}_2), \ldots, (L^{TG}_k, U^{TG}_k)\rangle$ where L^{TG}_i and L^{TG}_i are defined as follows:*

$$L^{TG}_i = \{(p_S, p_{\mathcal{O}}) \in Q^{TG} \mid q_{\mathcal{O}} \in L^{\mathcal{O}}_i\}$$
$$U^{TG}_i = \{(p_S, p_{\mathcal{O}}) \in Q^{TG} \mid q_{\mathcal{O}} \in U^{\mathcal{O}}_i\}$$

4. *The set of counters $\mathcal{C}^{TG}$ is directly inherited from $\mathcal{O}bs$:*

$$\mathcal{C}^{TG} = \mathcal{C}^{\mathcal{O}}$$

3.2 Test cases selection

The purpose of the test case selection is to generate a particular test case TC from the test graph TG. Roughly speaking, it consists in "extracting" a subgraph of TG that are controllable and containing a least a sequence of $\mathcal{L}(\mathcal{O})$.

Clearly, to belong to $\mathcal{L}(\mathcal{O})$, an execution sequence of $\mathcal{O}$ has to reach a cycle containing a state belonging to some distinguished set L_i° (for some i) of the pair table associated to $\mathcal{O}$. Conversely, any sequence of $\mathcal{O}$ not leading to a strongly connected component of $\mathcal{O}$ containing a state of L_i° cannot belong to $\mathcal{L}(\mathcal{O})$. Therefore, we first define on TG the predicate L2L (for *"leads to L"*), to denote the set of states leading to such a strongly connected component:

$$\text{L2L}\,(q) \equiv \exists (q_1, q_2, \omega_1, \omega_2, \omega_3).(q \overset{\omega_1}{\Longrightarrow}_{T^{TG}} q_1 \overset{\omega_2}{\Longrightarrow}_{T^{TG}} q_2 \overset{\omega_3}{\Longrightarrow}_{T^{TG}} q_1 \text{and } \exists i.\ q_2 \in L_i^{\circ})$$

We can now define a subset of relation T^{TG}, controllable, and containing at least a sequence of $\mathcal{L}(\mathcal{O})$. This subset, computed by the function *select* below, contains all non controllable transition of T^{TG} (labelled by an element of A_u), and at most one (randomly chosen) controllable transition of T^{TG} leading to a state of L2L when several such transitions exist from a given state of TG:

$$\text{select}\,(T^{TG}) = \{(p,a,q) \in T^{TG} \mid a \in A_u \text{ or}$$
$$a = \text{ one-of}\,(\{a_i \in A_c \mid p \xrightarrow{a_i}_{T^{TG}} q_i \text{ and L2L}\,(q_i)\})\}$$

Note that this function preserves the reachability of states belonging to $\mathcal{L}^{\mathcal{O}}$.

Finally, this subset of T^{TG} remains to be extended with all (non controllable) action of a_u not explicitly appearing in T^{TG}, to ensure that the test case execution will never be stopped by reception of an unexpected event. The definition of a test case TC is then the following:

Definition 6. *let $(TG, \mathcal{T}^{TG}, \mathcal{C}^{TG})$ a test graph with TG=$(Q^{TG}, A^{TG}, T^{TG}, q_{init}^{TG})$ and TA = (A_c, A_u) a test architecture. A test case $(TC, \mathcal{T}^{TC}, \mathcal{C}^{TC})$ is a Parameterised Rabin automaton with TC=$(Q^{TC}, A^{TC}, T^{TC}, q_{init}^{TC})$ such that $q_0^{TC} = q_0^{TG}$, $A^{TC} = A^{TG} \cup A_u$, Q^{TC} is the subset of Q^{TG} reachable by T^{TC} from q_0^{TC}, $\mathcal{T}^{TC}$ is the restriction of $\mathcal{T}^{TG}$ over Q^{TC}, and T^{TC} is defined as follows:*

$$T^{TC} = \textit{select}\,(T^{TG}) \cup \{(p,a,p) \mid a \in A_u \textit{ and} \nexists q.\ (p,a,q) \in T^{TG}\}$$

3.3 Implementation issues

We briefly sketch the concrete algorithms that could be used to implement the test case generation method proposed in this section. The objective here is not to provide a detailed implementation description (beyond the scope of this paper), but rather to give some indications on its algorithmic complexity. A possible (and simple) approach to compute a test case TC from a specification S, an observer $\mathcal{O}$bs and a test architecture TA is to proceed as follows:

1. computation of S (determinisation and suspension of S_0) and computation of sets $Q^{\otimes}$ and $T^{\otimes}$ introduced in definition 5. These operations can be done during a joint traversal of S and $\mathcal{O}$.
2. computation of the test graph TG (sets Q^{TG} and T^{TG}) from the previous result. This can be done through a single traversal of $Q^{\otimes}$ and $T^{\otimes}$.
3. computation of the strongly connected components of TG containing a distinguished state of L_i°, using for instance Tarjan's algorithm [18]. This operation also gives the L2L predicate.
4. test case selection (computation of function *select*) using a backward traversal of TG.

Apart the determinisation phase, all these operations remain linear in the number of transitions of the LTS considered, but the test graph has to be explicitly stored. However, some of the algorithms proposed in the TGV tool could certainly be used to perform most of these operations on an on-the-fly basis. This point has not been investigated at this time.

4 Example

4.1 System description.

We consider a control system for an automatic door, specified by the IOLTS given in figure 4. The behaviour of the controller is the following: it can receive a request for opening the door (`REQOPEN`), the door is then successively open (`OPEN`) and closed (`CLOSE`). It can also receive a `LOCK` request, those effect is to definitely lock the door, or any other requests `OTHER`, that are silently ignored. All these actions are supposed to belong to the test architecture. A possible specification of the controller is the IOLTS shown at the figure 4.

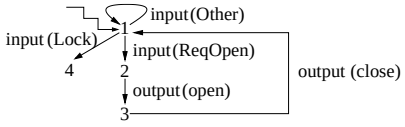

Fig. 4. Specification

The property we want to test on this system is: whenever the door is open, then it should be closed before a given amount of time (to be precised at test time). The negation of this property (the observer) is modelled by the Parameterised Rabin automaton of figure 5, where the α label denotes any observable action other than `CLOSE` (including δ). We now assume that the IUT is not quite conform to the specification. In particular it may spontaneously output an `ABORT` action and re-enter the initial state. The corresponding IOLTS is pictured on figure 8.

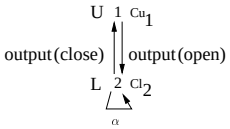

Fig. 5. Observer

4.2 Test Graph generation.

The first step consists in generating a test graph from the specification and the observer. The corresponding deterministic parameterised Rabin automaton is shown on figure 6. Note that the sets L and U are inherited from the observer. On this test graph, the execution sequences belonging to the language of the observer are the ones ending by α^ω (namely in states 32 and 42).

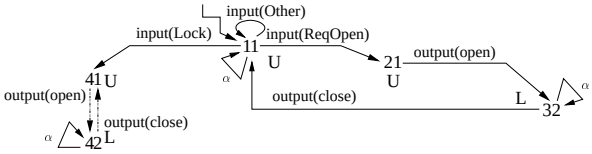

Fig. 6. Test Graph

4.3 Test Selection and test execution.

From the test graph we can then extract some particular test cases, for instance the one pictured on figure 7.

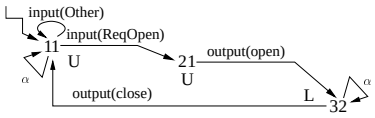

Fig. 7. Test Case

Transitions labelled with α indicate that the test case is output complete. Executing this test case may exhibit a possible incorrect behaviour of the IUT (figure 8), in which an occurrence of the `ABORT` action in state 32 leads to a **Fail** verdict (since the IUT is deadlocked in this state).

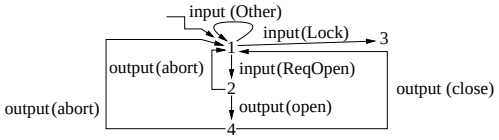

Fig. 8. IUT

More precisely, each time states 32, 11 or 21 are reached their respective counter are incremented. So during the test execution, the counter associated with the state 32 can overflow if an `ABORT` action occurred. Of course, this scenario is not guaranteed to appear since this incorrect behaviour is not fully controllable by the tester.

5 Conclusion

In this paper, we have proposed an approach to automatically produce test cases allowing to check the satisfiability of a linear property on a given implementation. Parameterised test cases are generated from a (possibly partial) specification of the IUT, the (bounded liveness) property being expressed itself by a Parameterised Rabin automaton. The resulting test case can then be instantiated only at test time, depending on the test environment considered (for instance the target architecture, or the actual communication structure between the tester and the IUT, etc.). This approach has been formally defined, and a practical test generation algorithm has been sketched.

The objective of this work is to extend to other contexts or application domains the framework of conformance testing, already well established in the telecommunication area. We believe that a prerequisite was to make this framework more flexible, for instance allowing the use partial specifications, or allowing the validation of explicit properties. This is a first step in this direction.

This work can now be extended in several directions. First we need to prototype the algorithms we have proposed to better estimate their performances, and possible optimisations. Then, their application on various case studies will certainly allow to improve the test selection strategy (possibly using TGV-like test purposes in combination with a property). Finally, the use of static analysis techniques (for instance as presented in [2]) could also certainly improve the efficiency of the test generation algorithm by focusing on most promising parts of the specification.

Acknowledgement. We would like to thank the people of a French group working on *robustness testing*, inside a French action supported by the Cnrs and gathering members of Irisa, Laas, Labri, Lri and Verimag laboratories

(http://www.laas.fr/TSF/AS23/). We would like also to thank the anonymous referees.

References

1. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation : a Simple Experiment. In *12th International Workshop on Testing of Communicating Systems*, G. Csopaki et S. Dibuz et K. Tarnay, 1999. Kluwer Academic Publishers.
2. M. Bozga, J.-C. Fernandez, and L. Ghirvu. Using static analysis to improve automatic test generation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 235–250, 2000.
3. E. Brinksma. A theory for the derivation of tests. In S. Aggarval and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII, IFIP*, pages 63–74. Elsevier Science Publishers, B.V., North-Holland, 1988.
4. J. Büchi. On a decision method in restricted second order arithmetic. In N. et al., editor, *Logic, Methodology and Philosophy of Sciences*. Stantford Univ. Press, 1962.
5. J.-. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*. LNCS 1102 Springer Verlag, 1996.
6. R. Groz, T. Jeron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montreal, Quebec*, pages 135–152, Elsevier, Juin 1999.
7. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992.
8. B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *In Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000),IEEE Computer Society Press*, pages 229–237, Grenoble, septembre 2000.
9. M. Mukund. Finite-state automata on infinite input. Technical report, 1996.
10. I. Parissis and J. Vassy. Test des proprietes de surete. In *In Actes du colloque Modelisation de Systemes Reactifs (MSR'01)*, pages 563–578, Hermes, 2001.
11. M. Phalippou. *Relations d'implantations et Hypothèses de Test sur des automates à entrées et sorties*. Thèse de doctorat, Université de Bordeaux, France, 1994.
12. M. Phalippou. Test sequence using Estelle or SDL structure information. In *FORTE'94*, Berne, Oct. 1994.
13. P.Thevenod-Fosse. Unit and integration testing of lustre programs: a case study from the nuclear industry. In *9th European Workshop on Dependable Computing (EWDC-9)*, pages 121–124, Gdansk, Pologne, 14-16 Mai 1998.
14. M. Rabin. *Automata o, Infinite Object and Church' Problem*. Number 13 in Regional Conference series in mathematics. American Mathematical Society, 1972.
15. P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, dec 1998.
16. S. Safra. On the complexity of ω-automata, checking. pages 319–327, 1988.
17. M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. Autolink - A Tool for Automatic and Semi-Automatic Test Generation from SDL Specifications. Technical Report A-98-05, Medical University of Lübeck, 1998.
18. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computation*, 2(1), june 1972.

19. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.
20. P. Wolper, M. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *24th IEEE Symposium of Foundations of Computer Science*, 1983.