Test Generation Derived from Model-Checking *

Thierry Jéron and Pierre Morel

IRISA / INRIA Rennes, Campus de Beaulieu, F-35042 Rennes, France,
{Thierry.Jeron, Pierre.Morel}@irisa.fr

Abstract. Model-checking and testing are different activities, at least conceptually. While model-checking consists in comparing two specifications at different abstraction levels, testing consists in trying to find errors or gain some confidence in the correctness of an implementation with respect to a specification by the execution of test cases. Nevertheless, there are also similarities in models and algorithms. We argue for this by giving a new on-the-fly test generation algorithm which is an adaptation of a classical graph algorithm which also serves as a basis of some model-checking algorithms. This algorithm is the Tarjan's algorithm which computes the strongly connected components of a digraph.

1 Introduction

Conformance testing aims at applying test cases to an implementation under test (IUT) in order to detect errors or increase ones confidence in the fact that the IUT is correct with respect to its specification. It is a black box testing: the source of the IUT is unknown but its behavior is known by its interactions with the environment. Conformance testing is applied in several domains and especially in the domain of protocols where its activity is standardized but not well formalized by [1]. [16] partly bridges this gap by defining a formal framework but does not instantiate it into a precise test generation algorithm.

Nevertheless, a lot of theoretical work has been done on test generation algorithms. Some syntactical methods exist but we will limit our discussion to semantical ones. Semantical methods can be divided into two classes which differ on the models, theories and algorithms. Techniques based on automata theory (see e.g. [20] for a survey) use Mealy machines (automata with each transition labelled with an input and an output) as models. They theoretically have a powerful fault coverage but make strong asumptions on specifications and IUT and are limited to small specifications. The other class of semantical techniques uses the model of labelled transition systems (LTS). They stem from fundamental studies on testing theory [8, 2, 5]. Originally defined for general LTS, their applicability was not clear. But they were the starting points for more realistic theories based on LTS with differentiated input and output transitions named $IOSM$, $IOTS$ or $IOLTS$ [25, 21]. The central point is a conformance relation relating specifications to correct implementations. These methods at least insure

* This work has been partially supported by the french action Forma.

unbias (only non conformant implementations can be rejected by a test case) and "theoretical" exhaustiveness (under some assumptions on implementations, all non conformant implementations can be rejected by a test suite).

In [12] we proposed a first on-the-fly test generation algorithm and we completed the picture in [17]. These algorithms have been implemented in our prototype tool TGV and gave good results on industrial experiments [13]. The main algorithm was based on a traversal of the synchronous product of a test purpose automaton with an $IOLTS$ representing the observable behavior of the specification. We thought that test cases should be acyclic in order to ensure the finiteness of their execution on the IUT, so the algorithm was cutting loops. But test practitioners and standardized test suite showed us that this was not always the case. It is the reason why we started to investigate a way for producing test cases with loops.

Some model-checking algorithms for CTL [6] or LTL [23], in particular local or on-the-fly ones also have to tackle with loops. Some of these algorithms (see e.g. [7, 26]) are adaptations of the classical Tarjan's algorithm which computes strongly connected components (SCCs) of a digraph during a depth first search (DFS). For on-the-fly model-checking, this algorithm has the advantage to provide a diagnostic sequence in the stack as soon as a violation of the property is detected. This facility has been used for test generation [10] as the negation of the checked property can be seen as a test selection criterion, i.e. a test purpose.

In our opinion, this is not sufficient as diagnostic sequences have to be further transformed into test sequences by taking into account output freedom of the specification, thus giving test sequences with possibly a lot of *Inconclusive* verdicts [1]. These verdicts should be reduced to the minimum in generating more adaptative test cases. We believe that test generation can benefit from model-checking algorithms but they need some adaptations to the testing framework. We present here such algorithms.

The paper is organized as follows. We first present in Section 2 the models used for test generation. Section 3 then gives an iterative formulation of the Tarjan's algorithm and present it as a framework for the derivation of all other algorithms presented in the paper. The three subsequent sections present instantiations of this framework for test generation. Section 4 describes an algorithm computing the subgraph of all sequences leading to *Accept* states of the test purpose and can be seen as a complete diagnosis for the CTL property $AG\neg Accept$ (or a complete explanation of $EF Accept$). In Section 5 we extract one test case from this subgraph. Section 6 improves the first algorithm for on-the-fly generation by anticipating operations of the second algorithm. Section 7 then describes how these algorithms are integrated into our tool TGV. We conclude with a comparison with other works and perspectives.

[1] *Inconclusive* verdicts are given to correct outputs of the IUT which do not lead to the satisfaction of the test purpose.

2 Conformance Testing

In this section we introduce the models used for test case generation and how they are used to describe specifications, implementations, test cases and test purposes. These models are based on the classical model of labelled transition systems with distinguished inputs and outputs. We report to [25] for a precise definition of the testing theory used.

Definition 1. *An IOLTS is an LTS $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ with Q^M a finite set of states, A^M a finite alphabet partitioned into three distinct sets $A^M = A_I^M \cup A_O^M \cup I^M$ where A_I^M and A_O^M are respectively inputs and outputs alphabets and I^M is an alphabet of unobservable, internal actions, $\rightarrow_M \subset Q^M \times A^M \times Q^M$ is the transition relation and q_0^M is the initial state.*

We will use the classical following notations of LTS for $IOLTS$.
Let $q, q', q_{(i)} \in Q^M, Q \subseteq Q^M, a_{(i)} \in A_I^M \cup A_O^M, \tau_{(i)} \in I^M$, and $\sigma \in (A_I^M \cup A_O^M)^*$.
$q \overset{\epsilon}{\Rightarrow}_M q' \equiv (q = q' \vee q \overset{\tau_1 \ldots \tau_n}{\rightarrow}_M q')$ and $q \overset{a}{\Rightarrow}_M q' \equiv \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow}_M q_1 \overset{a}{\rightarrow}_M q_2 \overset{\epsilon}{\Rightarrow}_M q'$ which generalizes in $q \overset{a_1 \ldots a_n}{\Rightarrow}_M q' \equiv \exists q_0, \ldots q_n : q = q_0 \overset{a_1}{\Rightarrow}_M q_1 \ldots \overset{a_n}{\Rightarrow}_M q_n = q'$. $Trace_M(q) \equiv \{\sigma | q \overset{\sigma}{\Rightarrow}_M\}$ and $Trace_M(M) = Trace_M(q_0^M)$.
We note q $\mathbf{after}_M$ $\sigma \equiv \{q' | q \overset{\sigma}{\Rightarrow}_M q'\}$ and Q $\mathbf{after}_M$ $\sigma \equiv \cup_{q \in Q} q$ $\mathbf{after}_M$ σ. We define $Out_M(q) \equiv \{a \in A_O^M | q \overset{a}{\Rightarrow}_M\}$ and $Out_M(Q) \equiv \{Out_M(q) | q \in Q\}$. We will not always distinguish between an $IOLTS$ and its initial state and note $M \Rightarrow_M$ instead of $q_0^M \Rightarrow_M$. We will omit the subscript $_M$ when it is clear from the context.

A specification is given in a formal description language (e.g. SDL, LOTOS or Estelle) which semantics allows to describe the behavior of the specification by an $IOLTS$ $S = (Q^S, A^S, \rightarrow^S, q_0^S)$ [2]. The $IOLTS$ S and intermediate $IOLTS$ defined from S are not effectively built but we need to define them for reasonning. As usually, we will assume that the behavior of the IUT can also be described by an $IOLTS$ which can never refuse an input: $IUT = (Q^{IUT}, A^{IUT}, \rightarrow^{IUT}, q_0^{IUT})$ with $A^{IUT} = A_I^{IUT} \cup A_O^{IUT} \cup I^{IUT}$ and $A_I^S \subseteq A_I^{IUT}$ and $A_O^S \subseteq A_O^{IUT}$. We use a conformance relation which says that an IUT conforms to S if and only if after a trace of S, outputs of the IUT are outputs of S:
IUT $\mathbf{ioconf}$ $S \iff \forall \sigma \in Trace(S), Out(IUT\ \mathbf{after}_{IUT}\ \sigma) \subseteq Out(S\ \mathbf{after}_S\ \sigma)$
For the sake of clarity, we took the definition of **ioconf** but all results also apply to **ioco** [25] which considers quiescence (i.e. deadlock and output quiescence) as an observable event. In [17] we also treat livelocks. The relation **ioco** is obtained from **ioconf** by adding loops labelled with a particular output δ to quiescent states of S and IUT.

In practice, test purposes are used as test selection criteria. We formalize them by automata i.e $IOLTS$ with selected marked states. A test purpose is a deterministic $IOLTS$ $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ equipped with two sets of sink states $Accept^{TP}$ which defines $Pass$ verdicts and $Reject^{TP}$ which allows to limit the exploration of the graph S. We suppose that $A^{TP} = A^S$ (this authorizes

[2] LOTOS does not distinguishes between inputs and outputs and a renaming is necessary for test generation.

actions of TP to be internal actions of S which is useful for testing in context) and TP is complete ($\forall q \in Q^{\mathrm{TP}}, a \in A^{\mathrm{TP}} q \xrightarrow{a}_{\mathrm{TP}}$).

The synchronous product of S and TP is an $IOLTS$ $SP = (Q^{\mathrm{SP}}, A^{\mathrm{SP}}, \rightarrow_{\mathrm{SP}}, q_0^{\mathrm{SP}})$ with $Q^{\mathrm{SP}} = Q^{\mathrm{S}} \times Q^{\mathrm{TP}}$, $A^{\mathrm{SP}} = A^{\mathrm{S}}$, $(p,q) \xrightarrow{a}_{\mathrm{SP}} (p', q') \iff p \xrightarrow{a}_{\mathrm{S}} p'$ and $q \xrightarrow{a}_{\mathrm{TP}} q'$, $q_0^{\mathrm{SP}} = (q_0^{\mathrm{S}}, q_0^{\mathrm{TP}})$. It can be understood as an automaton with sets of sink states defined by $Accept^{\mathrm{SP}} = Q^{\mathrm{S}} \times Accept^{\mathrm{TP}}$ and $Reject^{\mathrm{SP}} = Q^{\mathrm{S}} \times Reject^{\mathrm{TP}}$.

As test generation only considers the observable behavior of S, a first step is to replace in SP all internal actions by τ, to reduce τ actions (while adding δ loops for **ioco**) and to determinize the result.

This defines an $IOLTS$ $SP_{\mathrm{VIS}} = (Q^{\mathrm{VIS}}, A^{\mathrm{VIS}}, \rightarrow_{\mathrm{VIS}}, q_0^{\mathrm{VIS}})$ with $Q^{\mathrm{VIS}} \subseteq 2^{Q^{\mathrm{SP}}}$, $A^{\mathrm{VIS}} = A_{\mathrm{I}}^{\mathrm{VIS}} \cup A_{\mathrm{O}}^{\mathrm{VIS}}$ with $A_{\mathrm{O}}^{\mathrm{VIS}} = A_{\mathrm{O}}^{\mathrm{SP}}$ and $A_{\mathrm{I}}^{\mathrm{VIS}} = A_{\mathrm{I}}^{\mathrm{SP}}$, $q_0^{\mathrm{VIS}} = q_0^{\mathrm{SP}}$ $\textbf{after}_{\mathrm{SP}}$ ϵ, $\forall a \in A^{\mathrm{VIS}}, \forall P, P' \in Q^{\mathrm{VIS}}, P \xrightarrow{a}_{\mathrm{VIS}} P' \iff P' = P$ $\textbf{after}_{\mathrm{SP}}$ a. SP_{VIS} is equipped with sets of sink states defined by $Reject^{\mathrm{VIS}} = \{s \in Q^{\mathrm{VIS}} \mid s \cap Reject^{\mathrm{SP}} \neq \emptyset\}$ and $Accept^{\mathrm{VIS}} = \{s \in Q^{\mathrm{VIS}} \mid s \cap Accept^{\mathrm{SP}} \neq \emptyset\} \setminus Reject^{\mathrm{VIS}}$.

Test cases are constructed from the $IOLTS$ SP_{VIS}. Before going to that construction, we must define what are test cases. A test case is an $IOLTS$ $TC = (Q^{\mathrm{TC}}, A^{\mathrm{TC}}, \rightarrow_{\mathrm{TC}}, q_0^{\mathrm{TC}})$ with distinguished subsets of states $Pass$, $Inconc$, and a new state $fail$. TC should have the following properties:

1. $Q^{\mathrm{TC}} \subseteq Q^{\mathrm{VIS}} \cup \{fail\}$, and $q_0^{\mathrm{TC}} = q_0^{\mathrm{VIS}}$,
2. $A^{\mathrm{TC}} = A_{\mathrm{I}}^{\mathrm{TC}} \cup A_{\mathrm{O}}^{\mathrm{TC}}$ with $A_{\mathrm{O}}^{\mathrm{TC}} \subseteq A_{\mathrm{I}}^{\mathrm{VIS}}$ and $A_{\mathrm{I}}^{\mathrm{TC}} \subseteq A_{\mathrm{O}}^{\mathrm{I}}$ (mirror image and all possible outputs of I considered),
3. $Pass = Accept^{\mathrm{VIS}} \cap Q^{\mathrm{TC}}$, $Inconc \subseteq Q^{\mathrm{VIS}}$, $Pass$, $Inconc$ and $fail$ are sink states and every state of TC except $fail$ can reach either a $Pass$ or an $Inconc$ state, $fail$ and $Inconc$ states can be reached directly only by inputs,
4. $\forall q \in Q^{\mathrm{TC}}, \forall a \in A_{\mathrm{I}}^{\mathrm{TC}}, (\exists q' \in Inconc \cup \{fail\}, q \xrightarrow{a}_{\mathrm{TC}} q' \Rightarrow q \xrightarrow{*} Pass)$ and $(q \xrightarrow{a}_{\mathrm{TC}} fail \Rightarrow q \overset{a}{\not\rightarrow}_{\mathrm{VIS}})$,
5. $\forall q, q' \in Q^{\mathrm{TC}}, \forall a \in A^{\mathrm{TC}}, q \xrightarrow{a}_{\mathrm{TC}} q' \wedge q' \neq fail \Rightarrow q \xrightarrow{a}_{\mathrm{VIS}} q'$,
6. maximality: $\forall q \in Inconc, q \overset{*}{\not\rightarrow}_{\mathrm{VIS}} Accept$,
7. controllability: $\forall q \in Q^{\mathrm{TC}}, \forall a \in A_{\mathrm{O}}^{\mathrm{TC}}, q \xrightarrow{a}_{\mathrm{TC}} \Rightarrow$ $\forall b \neq a, q \overset{b}{\not\rightarrow}_{\mathrm{TC}} \wedge \forall q \in Q^{\mathrm{TC}}, (\exists a \in A_{\mathrm{I}}^{\mathrm{TC}}, q \xrightarrow{a}_{\mathrm{TC}} \Rightarrow \forall b \in A_{\mathrm{I}}^{\mathrm{TC}}, q \xrightarrow{a}_{\mathrm{TC}})$.

Some of these properties come from the definition of **ioconf** and ensure unbias (i.e. no correct implementation can be rejected by a test case). Some other properties such as the controllability condition come from test practice: a tester always controls its outputs. The maximality property ensures that no $Inconclusif$ verdict can be given in a state where a $Pass$ verdict could eventually be obtained later. In a state where an input is possible all possible outputs of an IUT are considered. This input completion allows us to consider transitions leading to $fail$ as implicit in the sequel. These properties do not uniquely identify a test case, thus we have to provide a constructive algorithm which ensures them. This is the subject of the following sections.

3 Tarjan's Algorithm as a Framework

In this section we present an iterative version of the algorithm "StrongConnect" [24] computing the SCCs of a given digraph $G = (Q^{G}, A^{G}, \rightarrow_{G}, q_0^{G})$ as a framework to derive other algorithms. First, we introduce all the notions and results used to describe this framework and following algorithms. Finally, we show the framework "StrongConnect" and its resulting graph.

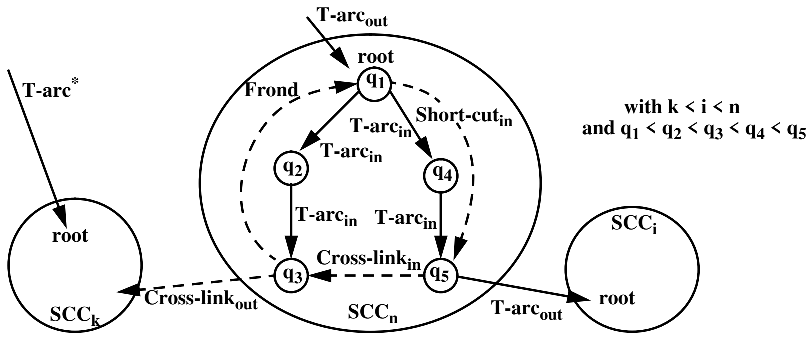

Fig. 1. Partition of edges defined by a DFS

Recall that a graph is strongly connected if for each pair of states (v,w) there is a path from v to v containing w. The SCCs of a graph G are the maximal strongly connected subgraphs of G. A DFS applied to G defines a spanning forest F by considering edges leading to unvisited states (tree-arcs). We suppose that states are numbered in the order in which they are reached during the search (field "number" of a state). Inspired from [24] a DFS defines a partition of edges (see Figure 1) : edges leading to a new state (not yet numbered) of the same (resp. distinct) SCC(s) are called "tree-arcs$_{in}$" (resp. "tree-arcs$_{out}$"); "fronds" are edges running from descendants to ancestors in the tree; "short-cuts$_{in}$" (resp. "short-cuts$_{out}$") are edges running from ancestors to descendants of the same (resp. distinct) SCC(s); edges in a SCC (resp. between two SCCs) running from one subtree to another subtree of F are called "cross-links$_{in}$" (resp. "cross-links$_{out}$"). For any frond (resp. short-cut) between two states v and u (resp. u and v), there exists a path of tree-arcs from u leading to v. A root of a SCC is the first reached state of this SCC and thus the smallest numbered state. The field "lowlink" of a state allows to detect the root of each SCC synthesizing the smallest state which is in the same component and is reachable by traversing zero or more tree-arcs followed by at most one frond or cross-link$_{in}$. A state is a root of a SCC if and only if its number and its lowlink are equal. The following framework is the well-known algorithm "StrongConnect" of Tarjan, where some sections (Start state, New state, Old state, State of a new SCC, Tree-arc backtrack) are left empty for derived algorithms. This algorithm identifies the set of SCCs of a graph G. The stack "Dfs_Stack" stores the current exploration sequence during the search and the stack "Scc_Stack" keeps all the visited states which SCC is still not completed. The field "act" of a state q is the label of the tree-arc leading to q.

The function Adjacency_Set gives fireable transitions from a given state q, so:
Adjacency_Set (q) := {(a, q') | (q, a, q') $\in \rightarrow_G$}
Procedure StrongConnect (**state :** q_{start});
state : q_{source}, q_{target}, q_{pred}, q; **adjacency_set** : Adj_{source}, Adj_{target}, Adj_{pred};
BEGIN
    Creation of *Dfs_Stack*; Creation of *Scc_Stack*;
    $q_{\text{start}}.number := q_{\text{start}}.lowlink := i := i + 1$; $q_{\text{start}}.act := \epsilon$;
    [Start state]
    Push (q_{start}, *Adjacency_Set*(q_{start})) in *Dfs_Stack*; Push q_{start} in *Scc_Stack*;
    while not empty *Dfs_Stack* **do begin**
        (q_{source}, Adj_{source}) := *Top*(*Dfs_Stack*);
        if not empty Adj_{source} **then begin**
            Remove (m, q_{target}) from Adj_{source};
            if q_{target} is not yet numbered **then begin** (*$(q_{\text{source}}, m, q_{\text{target}})$ is a tree-arc*)
                $q_{\text{target}}.number := q_{\text{target}}.lowlink := i := i + 1$; $q_{\text{target}}.act := m$;
                Push (q_{target}, *Adjacency_Set*(q_{target})) in *Dfs_Stack*; Push q_{target} in *Scc_Stack*;
                [New state]
            else begin (* $(q_{\text{source}}, m, q_{\text{target}})$ is not a tree-arc *)
                if $q_{\text{target}}.number < q_{\text{source}}.number$ **and** q_{target} in *Scc_Stack* **then**
                (* $(q_{\text{source}}, m, q_{\text{target}})$ is a frond or a cross-link$_{in}$ (in same SCC) *)
                    $q_{\text{source}}.lowlink := min(q_{\text{source}}.lowlink, q_{\text{target}}.number)$;
                [Old state]
            end
        else begin (* Adj_{source} is empty *)
            Pop (q_{source}) from *Dfs_Stack*;
            if $q_{\text{source}}.lowlink = q_{\text{source}}.number$ **then begin** (*q_{source} is root of SCC*)
                while $q := top(Scc\_Stack)$ satisfies $q.number \geq q_{\text{source}}.number$ **do**
                begin (* creation of a new component *)
                    Pop (q) from *Scc_Stack* and put q in current component;
                    [State of a new SCC]
                end
            end
            if not empty Dfs_Stack **then begin** (* backtracking *)
                $(q_{\text{pred}}, Adj_{\text{pred}}) := top(Dfs\_Stack)$;
                $q_{\text{pred}}.lowlink := min(q_{\text{pred}}.lowlink, q_{\text{source}}.lowlink)$;
                [Tree-arc backtrack]
            end
        end
    end
END;

Procedure MAIN (G);
BEGIN
  integer : i := 0;
  for q_{start} in Q^{G} **if** q_{start} not yet numbered **then** StrongConnect (q_{start});
END;

We can interpret the result of this program as the reduced directed acyclic graph (DAG) where a node is a SCC of G and where edges are either tree-arc$_{out}$, cross-link$_{out}$, or short-cut$_{out}$. This algorithm is linear in both space and time. Notice that if the input graph G is rooted, a call to StrongConnect with the root will visit all states.

4 Computation of the Complete Test Graph

Notice that if the controllability condition defined in Section 2 is suppressed and in condition 3 we redefine $Pass$ by $Pass = Accept_{\mathrm{VIS}}$, the resulting set of properties uniquely defines a subgraph of SP_{VIS} called *Complete Test Graph* (CTG) as it defines all potential test cases w.r.t. TP. Even if it does not define a test case, it is sometimes interesting to produce CTG and then to seperate it into a set of test cases. The following algorithm instantiating the undefined parts of StrongConnect framework computes the subgraph of the graph SP_{VIS} composed of all sequences leading to states in $Accept^{\mathrm{VIS}}$. Moreover, as **ioco** forces to take into account all outputs of the specification, those not leading to $Accept^{\mathrm{VIS}}$ have to be kept. Target states are put in the $Inconclusive$ set.

$$\begin{aligned} Trace(CTG) = &\{\sigma \mid SP_{\mathrm{VIS}}\ \textbf{after}\ \sigma \subseteq Accept^{\mathrm{VIS}}\} \\ &\cup \{\sigma.a \in Trace(SP_{\mathrm{VIS}}) \mid a \in A_{\mathrm{O}}^{\mathrm{VIS}} \wedge \exists \sigma' (SP_{\mathrm{VIS}}\ \textbf{after}\ \sigma.\sigma' \subseteq\ Accept^{\mathrm{VIS}})\} \end{aligned}$$

The problem of finding this subgraph reduces to the problem of finding the reduced DAG of SCCs which lead to an *Accept* state and thus reduces to find the roots of SCCs leading to an *Accept* state. For a DAG, a simple DFS allows to correctly synthesize the reachability to an *Accept* state along tree-arcs$_{out}$, cross-links$_{out}$ and short-cuts$_{out}$ (there is no other type of edge in such a graph). This property is used to prove the correctness of the synthesis for each root of SCC using the underlying DAG structure of SCCs. Notice that a short-cut$_{out}$ between u and v does not give additional information regarding reachability in u w.r.t. v because there exists another path of tree-arcs from u leading to v. A root of a SCC leads to an *Accept* state if and only if it is an *Accept* state or a state of its SCC can reach another SCC, by a tree-arc$_{out}$ or cross-link$_{out}$, which leads to an *Accept* state. The field "*L2A*" of a state meaning "Leads to an Accept state" is used to synthesize this reachability information. Moreover, *L2A* is also used for garbage collection of unnecessary parts of the graph.

When we reach a state for the first time, its $L2A$ field is initialized to true if and only if it is an *Accept* state. So:

[Start state]

$q_{\text{start}}.L2A := q_{\text{start}} \in Accept^{\text{VIS}}$;

if $q_{\text{start}} \in Reject^{\text{VIS}} \cup Accept^{\text{VIS}}$ **then**
remove all (q_{start}, a, q') from $\rightarrow_{\text{VIS}}$;

[New state]

$q_{\text{target}}.L2A := q_{\text{target}} \in Accept^{\text{VIS}}$;

if $q_{\text{target}} \in Reject^{\text{VIS}} \cup Accept^{\text{VIS}}$ **then**
remove all (q_{target}, a, q') from $\rightarrow_{\text{VIS}}$;

When a state is reached again, only cross-link$_{out}$ transitions add more information about reachability to an *Accept* state to the root of a strongly connected component. An input short-cut$_{out}$ or cross-link$_{out}$ to a SCC not leading to *Accept* is pruned. So:

[Old state]

if q_{target} not in Scc_Stack **then begin**
(* It is a short-cut$_{out}$ or cross-link$_{out}$*)
if $q_{\text{target}}.number < q_{\text{source}}.number$
then (* It is a cross-link$_{out}$ *)
$q_{\text{source}}.L2A := q_{\text{source}}.L2A \vee q_{\text{target}}.L2A$;
if $\neg q_{\text{target}}.L2A \wedge \text{m} \in A_{\text{I}}^{\text{VIS}}$ **then**
remove (q_{source}, m, q_{target}) from $\rightarrow_{\text{VIS}}$;
end

When a root of a SCC is found, its $L2A$ field is correct. All the states of this SCC update their $L2A$ field w.r.t. their root and the part of the graph which cannot lead to *Accept* is pruned.

[State of a new SCC]

$q.L2A := q_{\text{source}}.L2A$;

if $\neg q.L2A$ **then**
remove all (q, a, q') from $\rightarrow_{\text{VIS}}$;

We have to synthesize the reachability information along tree-arcs. An input tree-arc$_{out}$ leading to a state not in CTG is pruned. So:

[Tree-arc backtrack]

$q_{\text{pred}}.L2A := q_{\text{pred}}.L2A \vee q_{\text{source}}.L2A$;

if $q_{\text{source}}.number = q_{\text{source}}.lowlink \wedge \neg q_{\text{source}}.L2A \wedge m' := q_{\text{source}}.act \in A_{\text{I}}^{\text{VIS}}$
then
remove (q_{pred}, m', q_{source}) from $\rightarrow_{\text{VIS}}$;

Let CTG be the subgraph obtained by this algorithm from SP_{VIS} and reduced to the accessible part from its initial state. $CTG = (Q^{\text{CTG}}, A^{\text{CTG}}, \rightarrow_{\text{CTG}}, q_0^{\text{CTG}})$, with two sets of marked states $Pass^{\text{CTG}}$ and $Inconc^{\text{CTG}}$ such that:
$A^{\text{CTG}} = A_{\text{O}}^{\text{CTG}} \cup A_{\text{I}}^{\text{CTG}}$ with $A_{\text{O}}^{\text{CTG}} \subseteq A_{\text{I}}^{\text{VIS}}$ and $A_{\text{I}}^{\text{CTG}} = A_{\text{O}}^{\text{VIS}}$ (mirror image),
$\rightarrow_{\text{CTG}} = \{(v,a,w) \in \rightarrow_{\text{VIS}} \mid v.L2A \wedge (w.L2A \vee a \in A_{\text{I}}^{\text{CTG}})\}$, $q_0^{\text{CTG}} = q_0^{\text{VIS}}$,
$Q^{\text{CTG}} = \{v \in Q^{\text{VIS}} \mid q_0^{\text{CTG}} \xrightarrow{*}_{\text{CTG}} v\}$, $Pass^{\text{CTG}} = Accept^{\text{VIS}} \cap Q^{\text{CTG}}$,
$Inconc^{\text{CTG}} = \{v \in Q^{\text{CTG}} \backslash Pass^{\text{CTG}} \mid \forall a \in A^{\text{CTG}} \wedge w \in Q^{\text{CTG}}, (v,a,w) \notin \rightarrow_{\text{CTG}}\}$.

The graph CTG contains all the behaviors a test case might have. According to the test case definition, now we have to deal with controllability.

5 Extraction of a Controllable Test Graph

We present here an algorithm based on the StrongConnect framework and computing a subgraph of the accepted subgraph CTG, controllable and where each state can reach $Pass$ state or is an $Inconclusive$ state. Informally, the adaptation of StrongConnect consists in a DFS starting from each $Pass$ state of CTG and using the predecessor transition relation. A correction of possibly controllability conflicts is done for all new reached state by pruning conflicting actions with the current action. This may modify the reachability from the initial state which is synthesized in order to determine the set of states of the resulting controllable test case. This is based on the same scheme as the synthesis of $L2A$ (field "$rfis$" for "Reachable From the Initial State").

The function Adjacency_Set takes into account the backward sight of the new algorithm.

$Adjacency\_Set(q) :=$
$\{(a, q') | (q', a, q) \in \rightarrow_{\text{CTG}}\}$

The procedure Pruning allows to prune transitions of CTG causing controllability conflicts with the current transition of a given state.

Procedure Pruning (q, m)
Begin
if $m \in A_{\text{O}}^{\text{CTG}}$ **then** (* remove all others *)
$\forall\ m' \neq m$, remove (q, m', q') from $\rightarrow_{\text{CTG}}$;
else (* remove all the outputs *)
$\forall m' \in A_{\text{O}}^{\text{CTG}}$, remove (q, m', q') from $\rightarrow_{\text{CTG}}$;
End

The initialization and synthesis of the $rfis$ field are based on the $L2A$ field scheme. We prune conflicting transitions when a new state is reached.

[Start state]
$q_{\text{start}}.rfis := (q_{\text{start}} = q_0^{\text{CTG}})$;
[New state]
$q_{\text{target}}.rfis := (q_{\text{target}} = q_0^{\text{CTG}})$;
Pruning (q_{target}, m);
[Old State]
$q_{\text{source}}.rfis := q_{\text{source}}.rfis\ \vee\ q_{\text{target}}.rfis$;
if $\neg q_{\text{target}}.rfis \wedge m \in A_{\text{O}}^{\text{CTG}}$ **then**
remove $(q_{\text{target}}, m, q_{\text{source}})$ from $\rightarrow_{\text{CTG}}$
[State of a new SCC]
$q.rfis := q_{\text{source}}.rfis$;
if $\neg q.rfis$ **then**
remove all (q, a, q') from $\rightarrow_{\text{CTG}}$;
[Tree-arc backtrack]
$q_{\text{pred}}.rfis := q_{\text{pred}}.rfis\ \vee\ q_{\text{source}}.rfis$;
if $q_{\text{source}}.number = q_{\text{source}}.lowlink\ \wedge \neg q_{\text{source}}.rfis \wedge m' := q_{\text{source}}.act \in A_{\text{O}}^{\text{CTG}}$ **then**
remove $(q_{\text{source}}, m', q_{\text{pred}})$ from $\rightarrow_{\text{CTG}}$;

Let $TC = (Q^{\text{TC}}, A^{\text{TC}}, \rightarrow_{\text{TC}}, q_0^{\text{TC}})$, with two sets $Pass^{\text{TC}}$ and $Inconc^{\text{TC}}$ such that : $A^{\text{TC}} = A_{\text{O}}^{\text{TC}} \cup A_{\text{I}}^{\text{TC}}$ with $A_{\text{O}}^{\text{TC}} = A_{\text{O}}^{\text{CTG}}$ and $A_{\text{I}}^{\text{TC}} = A_{\text{I}}^{\text{CTG}}$,
$\rightarrow_{\text{TC}} = \{(v, a, w) \rightarrow_{\text{CTG}} | v.rfis\ \wedge\ (w.rfis\ \vee\ a \in A_{\text{I}}^{\text{TC}})\}$, $q_0^{\text{TC}} = q_0^{\text{CTG}}$,
$Q^{\text{TC}} = \{v \in Q^{\text{CTG}} \mid q_0^{\text{TC}} \overset{*}{\rightarrow}_{\text{TC}} v\}$, $Pass^{\text{TC}} = Pass^{\text{CTG}} \cap Q^{\text{TC}}$,
$Inconc^{\text{TC}} = \{v \in Q^{\text{TC}} \backslash Pass^{\text{TC}} \mid \forall a \in A^{\text{TC}}\ \wedge\ w \in Q^{\text{TC}}, (v, a, w) \notin \rightarrow_{\text{TC}}\}$.
Remark: The order in which we apply "controllable" StrongConnect to $Pass$ states of CTG influences the resulting test case. Breadth-first search starting from the set of $Pass$ states could give shortest test cases, but StrongConnect allows to interleave garbage collection. The graph CTG represents all the behavior to be tested w.r.t. a test purpose. We can derive from this graph sequential, arborescent, or looping test cases. We could even apply some automata based methods such as UIO to each SCC of CTG.

6 Solving Controllability Conflicts during Forward Search

In the previous section, we have shown an algorithm resolving all the controllability conflicts of the graph CTG. In fact some conflicts can be solved during the forward DFS as will be shown in this section.

Notice that in the complete algorithm, we attempt to synthesize in an efficient way the information of reachability to an *Accept* state on all the roots of SCC of SP_{VIS}. First, we prone that a controllability conflict in a state can be removed during this algorithm in the case where we know that this state leads to an *Accept* state. This is done while backtracking a transition m between a source state such that its $L2A$ is still false and a target state such that its $L2A$ is true. In this case, we can prune all the transitions from the source state in conflict with m. This assumption leads us to synthesize $L2A$ not only for a root of a SCC but also for all the states of this SCC which can get the information earlier and then to prune parts of the initial graph earlier and thus saving time. In this algorithm refined from the CTG computation, the synthesis of $L2A$ information is done along tree-arcs, cross-links and fronds and possible pruning actions are done earlier. As seen before, short-cut transitions give redundant information.

The function "pruning" not only prune conflicting already synthesized transitions but also conflicting transitions not already treated.

Procedure Pruning (q, m, Adj)
Begin
if $m \in A_I^{VIS}$ **then**
begin (*remove all other transitions*)
$Adj := \emptyset$;
$\forall m' \neq m$, remove all (q, m', q') from $\rightarrow_{VIS}$;
else begin (* remove all the inputs *)
$\forall m' \in A_I^{VIS}$, remove all (q, m', q') from $\rightarrow_{VIS}$
$\forall m' \in A_I^{VIS}$, remove all (m', q') from Adj;
end
End

[Start state], **[New state]** and **[State of a new SCC]** are identical to parts of the complete test graph computation. The synthesis of $L2A$ is done along cross-links, short-cuts, fronds and tree-arcs. Now, pruning actions are done earlier when backtracking to a state which knows that it leads to *Accept*.

[Old State]
if $q_{\text{target}}.L2A \wedge \neg q_{\text{source}}.L2A$ **then**
Pruning $(q_{\text{source}}, m, Adj_{\text{source}})$;
else if $q_{\text{target}} \notin Scc\_Stack \wedge \neg q_{\text{target}}.L2A \wedge m \in A_I^{VIS}$ **then**
remove $(q_{\text{source}}, m, q_{\text{target}})$ from $\rightarrow_{VIS}$;
$q_{\text{source}}.L2A := q_{\text{source}}.L2A \vee q_{\text{target}}.L2A$;
[Tree-arc backtrack]
if $q_{\text{source}}.L2A \wedge \neg q_{\text{pred}}.L2A$ **then**
Pruning $(q_{\text{pred}}, q_{\text{source}}.act, Adj_{\text{pred}})$;
else if $q_{\text{source}} \notin Scc\_Stack \wedge \neg q_{\text{source}}.L2A \wedge m' := q_{\text{source}}.act \in A_I^{VIS}$ **then**
remove $(q_{\text{pred}}, m', q_{\text{source}})$ from $\rightarrow_{VIS}$;
$q_{\text{pred}}.L2A := q_{\text{pred}}.L2A \vee q_{\text{source}}.L2A$;

Remark: The resulting graph is a subgraph of CTG. Some controllability conflicts persist in some states which have synthesized the $L2A$ information w.r.t. their root only. We have to apply to this resulting graph the algorithm of the previous section to correct persistent conflicts.

7 Tool

Architecture and Algorithms: The algorithms presented in the paper are the basis of our prototype tool TGV developed in collaboration with IRISA/INRIA Rennes and Verimag Grenoble [12, 13, 17, 4]. In Section 2 we have presented all $IOLTS$ considered for the test cases generation: S, TP, SP, SP_{VIS}. As TGV works on-the-fly, only necessary parts of these $IOLTS$ are constructed on demand in a lazy way. This imposes that $IOLTS$ are implicit and accessed through APIs giving the functions for their construction: the initial state, the transition relation and comparison of states. TGV also uses an API of CADP [11] to store parts of all intermediate $IOLTS$. TGV is interfaced with several different simulators which provide the API for S. In particular it is interfaced with the SDL simulator ObjectGeode from Verilog [3] and the LOTOS simulator from CADP.

Another central algorithm is the algorithm which computes SP_{VIS} from SP already presented in [17]. It combines several aspects: addition of δ actions in the case of quiescence, τ-reduction and determinization. All this is done on-the-fly with again an adaptation of StrongConnect interleaved with a classical subset construction for determinization (see e.g. [15]). In this case StrongConnect is applied to the subgraph of SP composed of τ-actions. Meanwhile, observable actions and target states are synthesized on top of the subgraphs and a subset construction is applied for determinization. StrongConnect starts from the initial state of SP and creates new initial states for subsequent calls to StrongConnect each time an observable action reaches a new state until no new initial state is created. Links from states to their SCCs are stored avoiding to explore an already computed SCC. The application of Tarjan algorithm has linear complexity in time and space but the subset construction involves an exponential blow up in the size of the resulting graph. Nevertheless as TGV is applied on-the-fly we have been able to tackle examples with a lot of internal actions (specifications describing services for example) where determinization was the bottleneck for methods with complete state graph generation.

Case Studies: The first experiment of TGV [13] was done on an SDL specification of an ISDN protocol named DREX. Even with this embryonic version implementing the algorithm of [12] which did not completely work on the fly, TGV proved its efficiency and the quality of generated test cases compared to manual ones. TGV now works on-the-fly with the algorithms presented here and has been experimented on two industrial size case studies. The first one is an SDL specification of the SSCOP protocol of the ATM which served for many other case studies. This study allowed us to combine static analysis techniques in prelude to test generation and to use TGV on a multi-process specification in an asynchronous environment [4]. The second one is a LOTOS specification of a cache coherency protocol of a multiprocessor architecture of Bull [18]. Produced test cases have been executed on the real architecture and improved the test practice in a domain to which it was not originally dedicated. For these two case studies, on-the-fly generation proved its utility as it was impossible to generate the complete state graphs.

Comparison with other Techniques: Compared to TGV, methods based on automata theory (see e.g. [20]) have serious drawbacks. They need the construction of complete state graphs which limits their use to small specifications. As in TGV, they need abstraction and reduction of internal actions, determinization and often minimization, but on the whole state graph of the specification. They need the construction of identifying sequences (UIO for example) and quite complex algorithms to build test cases while TGV is linear in this phase. Their advantage is the complete coverage of a fault model but, as other methods, this needs assumptions on the implementation such as fairness. Often determinism is required but this is not realistic. A test suite is a monolithic sequence and does not correspond to hand-written test cases. Observable non-determinism (possible responses of the implementation to an input) is not really taken into account because output divergence of the specification w.r.t. the test sequence directly leads to an *Inconclusive* verdict . Nevertheless, test suites written by hand sometimes have identifying sequences that automatic tools should be able to produce. These sequences do not identify states of the state graph but control states of the specification. An idea is to use these methods on more abstract specifications (only the control part) in order to generate test purposes for control state identification.

This leads us to the comparison with TVEDA. TGV and TVEDA are complementary tools. TVEDA can produce automatically test purposes that can be used by TGV. But generating test purposes automatically is in general not sufficient to cover all interesting behaviors. So users will still have to specify some test purposes by hand.

In some aspects, TGV seems similar to Samstag [14] which uses test purposes specified by MSCs. But there are important differences. The first one is that the theory underlying Samstag is not clear. Nothing refers to any conformance relation or fault model which prevent from any proof on the correctness of generated test cases. Non-determinism is not taken into account because if a test purpose MSC does not define a "unique observable", it is rejected. A test purpose specified by an MSC must describe a complete sequence of observable events which makes it difficult to write and prevents for any abstraction. Finally, the algorithm is almost limited to checking that the MSC describes a (non-deterministic) behavior of the specification and completing the MSC with inputs leading to *Inconclusive*.

Trojka [9] has common points with TGV. It is based on the same theoretical background [25]. It performs on-the-fly test case generation in the sense that it can simultaneously execute them on the IUT. Trojka does not use test purposes as TGV but randomly chooses outputs of the test case among possible ones and checks the validity of inputs according to the observable behavior of the specification. This necessitates a function similar to τ-reduction and determinization . This has been implemented by a breadth traversal which prevents the detection of livelocks and may duplicate some work, problems which are solved by TGV with the computation of SCCs.

8 Conclusion

We have presented a new on-the-fly test case generation algorithm based on a classical graph algorithm also used in some local and on-the-fly model-checkers. This algorithm has complexity linear in the size of the observable behavior of the product of a test purpose and a specification. It produces test cases of high quality and very similar to those written by hand with choices and loops. This algorithm and those sketched in Section 7 are being transfered into the Object-

Geode tool from Verilog [3]. They will serve as the test generation engine which will accept test purposes either written by hand or obtained by simulation or automatically computed by a method derived from TVeda with a coverage strategy [19]. However, what is lacking in TGV is a clever treatment of data. For the moment, the stress has been put on control and data values are enumerated by the underlying simulation tools which may lead to a state explosion for specifications with large value domains. But we have started to study the possibility to combine the algorithms of TGV with a constraint solver and abstract interpretation techniques with the ambition to generate symbolic test cases with parameters and variables. Some ideas from previous works on TVEDA for example [22] could also be helpful.

References

1. ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework, 1992.
2. S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3), 1987.
3. B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project : A VALidatiON Environment For SDL/MSC Descriptions. In *6th SDL Forum, Darmstadt*, 1993.
4. M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, Special Issue on The Application of Formal Methods in Industrial Critical Systems*, To appear, 1999.
5. E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification VIII*, pages 63–74. North-Holland, 1988.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Languages and Systems*, 2(8):244–263, 1986.
7. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Workshop on Computer Aided Verification*. LNCS 531, June 1990.
8. R. De Nicola and M. Henessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
9. R.G. De Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2 1998.
10. A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Third Workshop TACAS, Enschede, The Netherlands*, LNCS 1217. Springer-Verlag, 1997.
11. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of CAV'96 (New Brunswick, New Jersey, USA)*. LNCS 1102, August 1996.

12. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In R. Alur and T.A. Henzinger, editors, *Proc. of CAV'96, (New Brunswick, New Jersey, USA)*. LNCS 1102, August 1996.
13. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocoles with Verification Technology. *Science of Computer Programming*, 29, 1997.
14. J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In O. Færgemand and A. Sarma, editors, *6th SDL Forum*, pages 253–266, Darmstadt (Germany), 1993. Elsevier Science B.V. (North-Holland).
15. J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley series in computer science, 1979.
16. ITU-T SG 10/Q.8 ISO/IEC JTC1/SC21 WG7. Information retrieval, transfer and management for OSI; framework: Formal Methods in Conformance Testing. CD 13245-1, ITU-T proposed recommendation Z500. ISO/ITU-T, 1996.
17. T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée: application à la génération de test. In G. Leduc, editor, *CFIP'97 : Ingénierie des Protocoles*. Hermes, September 1997.
18. H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11^{th} International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.
19. A. Kerbrat, T. Jéron, and R. Groz. Automated Test Generation from SDL Specifications. In *Proc. 9^{th} SDL FORUM (Montréal, Quebec, Canada)*, June 1999.
20. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84(8):1090–1123, August 1996.
21. M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.
22. M. Phalippou. Test sequence generation using Estelle or SDL structure information. In *FORTE'94*, Berne, October 1994.
23. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symposium on the Foundations of Computer Science*. ACM, November 1977.
24. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal Computing*, 1(2):146–160, June 1972.
25. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
26. B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In E. Best, editor, *CONCUR '93*, LNCS 715, pages 447–461. Springer-Verlag, 1993.