Test generation from recursive tiles systems

Sébastien Chédor, Thierry Jéron, Christophe Morvan

Test generation from recursive tiles systems

Sébastien Chédor[1], Thierry Jéron[2], Christophe Morvan[3]

[1] Université de Rennes I
[2] INRIA Rennes - Bretagne Atlantique,
[3] Université Paris-Est, Marne-La-Vallée, France
{sebastien.chedor, thierry.jeron}@inria.fr,
christophe.morvan@univ-paris-est.fr

Abstract. In this paper we explore test generation for *Recursive Tiles Systems* (RTS) in the framework of the classical **ioco** testing theory. The RTS model allows the description of reactive systems with recursion, and is very similar to other models like Pushdown Automata, Hyperedge Replacement Grammars or Recursive State Machines. We first present an off-line test generation algorithm for *Weighted* RTS, a determinizable sub-class of RTS, and second, an on-line test generation algorithm for the full RTS model. Both algorithms use test purposes to guide test selection through targeted behaviours.

1 Introduction and motivation

Conformance testing is the problem of checking by test experiments that a black-box implementation behaves correctly with respect to its specification. It is well known that testing is the most used validation technique to assess the quality of software systems, and represents the largest part in the cost of software development. Automatising is thus required in order to improve the cost and quality of the testing process. In particular, it is undoubtedly interesting to automate the test generation phase from specifications of the system. Formal model-based testing aims at resolving this problem by the formal description of testing artefacts (specifications, possible implementations, test cases) by mathematical models, formal definitions of conformance, the execution of tests and their verdicts, and the proof of some essential properties of test cases relating verdicts produced by test executions on implementations and conformance of these implementations with respect to their specifications. The **ioco** conformance theory introduced in [13] is a well established framework for the formal modelling of conformance testing for Input/Output Transition Systems (IOLTSs). Test generation algorithms and tools have been designed for this model [9,12] and for more general models whose semantics can be expressed in the form of infinite state IOLTSs [10,8].

In this paper, we are interested in test generation for reactive recursive programs, like the one in Fig 1. There already exist several ways to define recursive behaviours: pushdown automata (PDA), recursive state machines [1], regulars graphs, defined by functional (or deterministic) hyperedge replacement grammars (HR-grammars), [7,3]. Each of these models has its merits and flaws: PDA are classical, and well understood; recursive state machines are equally expressive and more visual as a model; HR-grammars are a visual model which characterizes the same languages but enables

```
static void main(String [] args){
   try{
      // Block 1 (input)
      int k =in.readInt();
      comp(k);
      // Block 2 (output)
      System.out.println("Done");
   }
   catch (Exception e){
      // Block 3 (output)
      System.out.println(e.getMessage());
   }
}
void comp (int x){
   // Block 4 (input)
   int res =1;
   boolean cont=in.readBoolean();
   if (cont){
      if (x==0)throw new Exception("An error occurred");
      // Block 5 (internal)
      res=x*comp(x-1);
      // Block 6 (output)
      System.out.println("Some text");
      return res;
   }
   else {
      // Block 7 (output)
      system.out.println("You stopped");
      return res;
   }
}
```

Fig. 1. A recursive program

to model systems having states of infinite degree. Furthermore, recent results define classes of such systems which may be determinized [5], which is of interest for test generation. The HR-grammars, on the other hand, are very technical to define. Here we try to get the best of both worlds: we use HR-grammars presented as tiling systems, called RTS (RTS). Such systems are mostly finite sets of finite LTS with frontiers, crossing the frontier corresponds to entering a new copy of one of the finite LTS. The semantics of an RTS is then an infinite state LTS. Hopefully for such models (co)-reachability which is essential for test generation using test purposes is decidable. Also determinization is possible for the class of *Weighted* RTS, which permits to design off-line test generation algorithms for this sub-class. For the whole class of RTS however determinization is impossible, but on-line test generation is still possible as subset construction is performed along finite executions.

To the best of our knowledge test generation for recursive programs has been seldom considered in the literature. The only work we are aware of is [6] which considers a model of deterministic PDA with inputs/outputs (IOPDS) and generate test cases in the same model. The present work can be seen as an extension of this, where non-determinism is taken into account.

Contribution and outline: The contribution of the paper is as follows. Section 2 recalls the main ingredients of the **ioco** testing theory for IOLTSs. In Section 3, we define the model of RTS for the description of recursive reactive programs, give its semantics in terms of an infinite state IOLTS obtained by recursive expansion of tiles. In Section 4, in the **ioco** framework, we propose an off-line test selection algorithm guided by test purposes for *Weighted* RTSs, a determinizable sub-class of RTSs, and prove essential properties of generated test cases. Furthermore in Section 5, we design an on-line test generation algorithm for the full RTS model, also using test purposes for test selection.

2 Conformance testing theory for IOLTS

This section recalls the **ioco** testing theory for the model of Input/Output Labelled Transition Systems that will serve as a basis for test generation from RTS. We first give a non-standard definition of IOLTS and introduce notations and basic operations, then review the **ioco** testing theory.

Definition 1 *An IOLTS (Input Output Labelled Transition System) is a tuple* $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \texttt{init}_{\mathcal{M}})$ *where* $Q_{\mathcal{M}}$ *is a set of* states*;* $\Sigma_{\mathcal{M}}$ *is the* alphabet of actions *partitioned into a set of inputs* $\Sigma^{?}_{\mathcal{M}}$*, a set of outputs* $\Sigma^{!}_{\mathcal{M}}$ *and a set of internal actions* $\Sigma^{\tau}_{\mathcal{M}}$ *and we denote by* $\Sigma^{o}_{\mathcal{M}} \triangleq \Sigma^{?}_{\mathcal{M}} \cup \Sigma^{!}_{\mathcal{M}}$ *the set of* visible actions [4]*;* $\Lambda_{\mathcal{M}}$ *is a set of* colours *with* $\texttt{init}_{\mathcal{M}} \in \Lambda_{\mathcal{M}}$ *a colour for initial states;* $\rightarrow_{\mathcal{M}} \subseteq Q_{\mathcal{M}} \times \Sigma_{\mathcal{M}} \times Q_{\mathcal{M}}$ *is the* transition relation*;* $\mathcal{C}_{\mathcal{M}} \subseteq Q_{\mathcal{M}} \times \Lambda_{\mathcal{M}}$ *is a relation between colours and states.*

In this non-standard definition of IOLTSs, colours are used to mark states by the relation $\mathcal{C}_{\mathcal{M}}$. For a colour $\lambda \in \Lambda_{\mathcal{M}}$, $\mathcal{C}_{\mathcal{M}}(\lambda) \triangleq \{q \in Q_{\mathcal{M}} \mid (q,\lambda) \in \mathcal{C}_{\mathcal{M}}\}$ and $\overline{\mathcal{C}_{\mathcal{M}}}(\lambda) \triangleq \{q \in Q_{\mathcal{M}} \mid (q,\lambda) \notin \mathcal{C}_{\mathcal{M}}\}$ denote respectively the sets of states coloured and not coloured by λ. In particular, $\mathcal{C}_{\mathcal{M}}(\texttt{init}_{\mathcal{M}})$ defines the set of initial states.

We write $q \xrightarrow{a}_{\mathcal{M}} q'$ for $(q,a,q') \in \rightarrow_{\mathcal{M}}$ and $q \xrightarrow{a}_{\mathcal{M}}$ for $\exists q' : q \xrightarrow{a}_{\mathcal{M}} q'$. This notation is generalized to sequences of actions, and for $w = \mu_1 \dots \mu_n \in (\Sigma_{\mathcal{M}})^*$, we note $q \xrightarrow{w}_{\mathcal{M}} q'$ for $\exists q_0, \dots, q_n : q = q_0 \xrightarrow{\mu_1}_{\mathcal{M}} q_1 \xrightarrow{\mu_2}_{\mathcal{M}} \cdots \xrightarrow{\mu_n}_{\mathcal{M}} q_n = q'$.

For $X \subseteq Q_{\mathcal{M}}$ a subset of states and $\Sigma' \subseteq \Sigma$ a sub-alphabet, we denote by $post_{\mathcal{M}}(\Sigma', X) = \{q' \in Q_{\mathcal{M}} \mid \exists q \in X, \exists \mu \in \Sigma' : q \xrightarrow{\mu}_{\mathcal{M}} q'\}$ the set of direct successors of a state in X by an action in Σ', and $pre_{\mathcal{M}}(\Sigma', X) = \{q \in Q_{\mathcal{M}} \mid \exists q' \in X, \exists \mu \in \Sigma' : q \xrightarrow{\mu}_{\mathcal{M}} q'\}$ the set of direct predecessors of X by a transition in Σ'. The set of states *reachable* from $P \subseteq Q_{\mathcal{M}}$ by actions in Σ' is $reach_{\mathcal{M}}(\Sigma', P) \triangleq \text{lfp}(\lambda X.P \cup post_{\mathcal{M}}(\Sigma', X))$ where lfp is the least fixed point operator. Similarly, the set of states *coreachable* from $P \subseteq Q_{\mathcal{M}}$ (*i.e.* the set of states from which P is reachable) is $coreach_{\mathcal{M}}(\Sigma', P) \triangleq \text{lfp}(\lambda X.P \cup pre_{\mathcal{M}}(\Sigma', X))$. We will also write $reach_{\mathcal{M}}(\Sigma', \lambda)$ for $reach_{\mathcal{M}}(\Sigma', \mathcal{C}_{\mathcal{M}}(\lambda))$ and $coreach_{\mathcal{M}}(\Sigma', \lambda)$ for $coreach_{\mathcal{M}}(\Sigma', \mathcal{C}_{\mathcal{M}}(\lambda))$.

$\Gamma_{\mathcal{M}}(q) \triangleq \{\mu \in \Sigma_{\mathcal{M}} \mid q \xrightarrow{\mu}_{\mathcal{M}}\}$ denotes the subset of actions enabled in q and respectively, $Out_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^{!}_{\mathcal{M}}$ and $In_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^{?}_{\mathcal{M}}$ denote the set of outputs (resp. inputs) enabled in q. For $P \subseteq Q_{\mathcal{M}}$, $Out_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} Out_{\mathcal{M}}(q)$ and $In_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} In_{\mathcal{M}}(q)$.

[4] In the examples, for readability reasons, we write $?a$ for an input $a \in \Sigma^{?}_{\mathcal{M}}$, $!x$ for an output $x \in \Sigma^{!}_{\mathcal{M}}$ and internal actions have no sign.

Visible behaviours of $\mathcal{M}$ are defined by the relation $\Longrightarrow_{\mathcal{M}} \in Q_{\mathcal{M}} \times (\{\epsilon\} \cup \Sigma^o_{\mathcal{M}}) \times Q_{\mathcal{M}}$ as follows: $q \stackrel{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q' \triangleq q = q'$ or $q \stackrel{\tau_1.\tau_2.\cdots\tau_n}{\longrightarrow}_{\mathcal{M}}{}^{*} q'$ and for $a \in \Sigma^o_{\mathcal{M}}$, $q \stackrel{a}{\Longrightarrow}_{\mathcal{M}} q' \triangleq \exists q_1, q_2 : q \stackrel{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q_1 \stackrel{a}{\longrightarrow}_{\mathcal{M}} q_2 \stackrel{\varepsilon}{\Longrightarrow}_{\mathcal{M}} q'$. For $\sigma = a_1 \cdots a_n \in (\Sigma^o_{\mathcal{M}})^*$ a sequence of visible actions, $q \stackrel{\sigma}{\Longrightarrow}_{\mathcal{M}} q'$ stands for $\exists q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Longrightarrow}_{\mathcal{M}} q_1 \cdots \stackrel{a_n}{\Longrightarrow}_{\mathcal{M}} q_n = q'$ and $q \stackrel{\sigma}{\Longrightarrow}_{\mathcal{M}}$ for $\exists\, q' : q \stackrel{\sigma}{\Longrightarrow}_{\mathcal{M}} q'$. We denote $q \ \texttt{after}\ \sigma \triangleq \{q' \in Q \mid q \stackrel{\sigma}{\Longrightarrow}_{\mathcal{M}} q'\}$ for the set of states in which one can be after observing σ starting from q and for $P \subseteq Q_{\mathcal{M}}$, $P \ \texttt{after}\ \sigma \triangleq \bigcup_{q \in P} q \ \texttt{after}\ \sigma$. $\mathsf{Traces}(q) \triangleq \{\sigma \in (\Sigma^o_{\mathcal{M}})^* \mid q \stackrel{\sigma}{\Longrightarrow}_{\mathcal{M}}\}$ denotes the set of sequences of visible actions that may be observed from q and $\mathsf{Traces}(\mathcal{M}) \triangleq \bigcup_{q_0 \in \mathcal{C}(\texttt{init}_{\mathcal{M}})} \mathsf{Traces}(q_0)$. $\mathsf{Traces}_P(\mathcal{M}) = \{\sigma \in (\Sigma^o_{\mathcal{M}})^* \mid (\mathcal{C}_{\mathcal{M}}(\texttt{init}_{\mathcal{M}})\ \texttt{after}\,\sigma) \cap P \neq \emptyset\}$ denotes the set of traces of sequences accepted in P.

$\mathcal{M}$ is *input-complete* if in each state all inputs are enabled, possibly after internal actions, i.e. $\forall q \in Q_{\mathcal{M}}, \forall a \in \Sigma^?_{\mathcal{M}}, q \stackrel{a}{\Longrightarrow}_{\mathcal{M}}$. $\mathcal{M}$ is *complete in a state* q if any action is enabled in q: $\forall q \in Q_{\mathcal{M}}, \Gamma(q) = \Sigma_{\mathcal{M}}$. $\mathcal{M}$ is *complete* if it is complete in all states.

An IOLTS $\mathcal{M}$ is deterministic if $|\mathcal{C}(\texttt{init}_{\mathcal{M}})| = 1$ (*i.e.* there is a unique initial state) and $\forall q \in Q_{\mathcal{M}}, \forall a \in \Sigma^o_{\mathcal{M}}, |q \ \texttt{after}\, a| \leq 1$, where $|.|$ is the cardinal of a set.

From an IOLTS $\mathcal{M}$, one can define a deterministic IOLTS $\mathcal{D}(\mathcal{M})$ with same traces as $\mathcal{M}$ as follows: $\mathcal{D}(\mathcal{M}) = (2^{Q_{\mathcal{M}}}, \Sigma^o_{\mathcal{M}}, \Lambda_{\mathcal{D}}, \rightarrow_{\mathcal{D}}, \mathcal{C}_{\mathcal{D}}, \texttt{init}_{\mathcal{D}})$ where for $P, P' \in 2^{Q_{\mathcal{M}}}$, $a \in \Sigma^o_{\mathcal{M}}$, $P \stackrel{a}{\longrightarrow}_{\mathcal{D}} P' \iff P' = P \ \texttt{after}\, a$, and $init_{\mathcal{D}} \in \Lambda_{\mathcal{D}}$ is the colour for the singleton state $\mathcal{C}_{\mathcal{D}}(init_{\mathcal{D}}) = \mathcal{C}_{\mathcal{M}}(\texttt{init}_{\mathcal{M}})\ \texttt{after}\,\varepsilon \in 2^{Q_{\mathcal{M}}}$. One can define other colours in $\Lambda_{\mathcal{D}}$ and, depending on the objective, the colouring $\mathcal{C}_{\mathcal{D}}$ may be defined according to $\Lambda_{\mathcal{M}}$ and $\mathcal{C}_{\mathcal{M}}$. For example, if $f \in \Lambda_{\mathcal{M}}$ defines marked states in $\mathcal{M}$, one may define a colour $F \in \Lambda_{\mathcal{D}}$ for $\mathcal{D}(\mathcal{M})$ such that $\mathsf{Traces}_{\mathcal{C}_{\mathcal{M}}(f)}(\mathcal{M}) = \mathsf{Traces}_{\mathcal{C}_{\mathcal{D}}(F)}(\mathcal{D}(M))$ simply by colouring by F the states in $s \in 2^{Q_{\mathcal{M}}}$ such that $\mathcal{C}(f)$ intersects s, *i.e.* at least one state in s is marked by f. Observe that the definition of $\mathcal{D}(\mathcal{M})$ is not always effective. However, it is the case whenever $\mathcal{M}$ is a finite state IOLTS. Even when it is effective, such a transformation may lead to an exponential blow-up. Often, for efficiency reasons, the full construction of $\mathcal{D}(\mathcal{M})$ is avoided, and on-the-fly paths are computed (visiting only a limited part of the powerset).

Synchronous product of IOLTS: One may define a product of two IOLTS such that sequences of actions in the product are the sequences of actions of both IOLTS:

Definition 2 *Let $\mathcal{M}_i = (Q_{\mathcal{M}_i}, \Sigma, \Lambda_{\mathcal{M}_i}, \rightarrow_{\mathcal{M}_i}, \mathcal{C}_{\mathcal{M}_i}, \texttt{init}_{\mathcal{M}_i})$, $i = 1, 2$ be two IOLTSs with same alphabet Σ. Their synchronous product $\mathcal{M}_1 \times M_2$ is the IOLTS $\mathcal{P} = (Q_{\mathcal{P}}, \Sigma_{\mathcal{P}}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, \mathcal{C}_{\mathcal{P}}, \texttt{init}_{\mathcal{P}})$ such that $Q_{\mathcal{P}} \triangleq Q_{\mathcal{M}_1} \times Q_{\mathcal{M}_2}$, and $\forall (q_1, q_2), (q'_1, q'_2) \in Q_{\mathcal{P}}$, $(q_1, q_2) \stackrel{a}{\longrightarrow}_{\mathcal{P}} (q'_1, q'_2) \triangleq q_1 \stackrel{a}{\longrightarrow}_{\mathcal{M}_1} q'_1 \wedge q_2 \stackrel{a}{\longrightarrow}_{\mathcal{M}_2} q'_2$. We define $\Lambda_{\mathcal{P}} \triangleq \Lambda_{\mathcal{M}} \times \Lambda_{\mathcal{M}'}$, in particular $\texttt{init}_{\mathcal{P}} \triangleq (\texttt{init}_{\mathcal{M}_1}, \texttt{init}_{\mathcal{M}_2})$, and for any $(\lambda_1, \lambda_2) \in \Lambda_{\mathcal{P}}$ the colouring relation is defined by $\mathcal{C}_{\mathcal{P}}((\lambda_1, \lambda_2)) \triangleq \mathcal{C}_{\mathcal{M}_1}(\lambda_1) \times \mathcal{C}_{\mathcal{M}_2}(\lambda_2)$.*

Specification and implementation: In the **ioco** testing framework, we assume that the behaviour of the specification is modelled by IOLTS $\mathcal{S} = (Q_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}}, \texttt{init}_{\mathcal{S}})$. The implementation under test is a black box system with same observable interface as the specification. In order to formalize conformance, it is usually assumed that the implementation behaviour can be modelled by an (unknown) input-complete IOLTS $\mathcal{I} = (Q_{\mathcal{I}}, \Sigma_{\mathcal{I}}, \Lambda_{\mathcal{I}}, \rightarrow_{\mathcal{I}}, \texttt{init}_{\mathcal{I}})$ with $\Sigma_{\mathcal{I}} = \Sigma^?_{\mathcal{I}} \cup \Sigma^!_{\mathcal{I}} \cup \Sigma^\tau_{\mathcal{I}}$ and $\Sigma^?_{\mathcal{I}} = \Sigma^?_{\mathcal{S}}$ and $\Sigma^!_{\mathcal{I}} = \Sigma^!_{\mathcal{S}}$.

Quiescence: It is current practice that tests observe traces of the implementation, and also absence of reaction (quiescence) using *timers*. Tests should then distinguish between quiescences allowed or not by the specification. Several kinds of quiescence may happen in an IOLTS: a state q is *output quiescent* if it is only waiting for inputs from the environment, *i.e.* $\Gamma(q) \subseteq \Sigma^?_{\mathcal{M}}$, (a *deadlock i.e.* $\Gamma(q) = \emptyset$ is a special case of output quiescence), and a *livelock* if an infinite sequence of internal actions is enabled, *i.e.* $\forall n \in \mathbb{N}, \exists \sigma \in (\Sigma^\tau_{\mathcal{M}})^n, q \xrightarrow{\sigma}_{\mathcal{M}}$ [5]. We note $quiescent(q)$ if q is either an output quiescence or in a livelock. From an IOLTS $\mathcal{M}$ one can build a new IOLTS $\Delta(\mathcal{M})$ where quiescence is made explicit by a new output δ:

Definition 3 *Let $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \texttt{init}_{\mathcal{M}})$ be an IOLTS, $\Delta(\mathcal{M})$ is the IOLTS $\Delta(\mathcal{M}) = (Q_{\mathcal{M}}, \Sigma_{\Delta(\mathcal{M})}, \Lambda_{\mathcal{M}}, \rightarrow_{\Delta(\mathcal{M})}, \mathcal{C}_{\mathcal{M}}, \texttt{init}_{\mathcal{M}})$ where $\Sigma_{\Delta(\mathcal{M})} = \Sigma_{\mathcal{M}} \cup \{\delta\}$ with $\delta \in \Sigma^!_{\Delta(\mathcal{M})}$ (δ is considered as an output, observable by the environment), and $\rightarrow_{\Delta(\mathcal{M})} = \rightarrow_{\mathcal{M}} \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$ is obtained from $\rightarrow_M$ by adding δ loops for each quiescent state q.*

In the sequel, we note $\Sigma^{!\delta}_{\mathcal{M}}$ for $\Sigma^!_{\mathcal{M}} \cup \{\delta\}$ and $\Sigma^{o\delta}_{\mathcal{M}}$ for $\Sigma^o_{\mathcal{M}} \cup \{\delta\}$. The traces of $\Delta(\mathcal{M})$ denoted by $\mathsf{STraces}(\mathcal{M})$ are called the *suspension traces* of $\mathcal{M}$. They represent the visible behaviour of $\mathcal{M}$, including quiescence and are the basis for the definition of the **ioco** conformance relation.

Conformance relation: In the **ioco** formal conformance theory [13], the implementation $\mathcal{I}$ conforms to its specification $\mathcal{S}$ if after any suspension trace σ of $\mathcal{S}$ the implementation $\mathcal{I}$ exhibits only outputs and quiescences that are specified in S. Formally:

Definition 4 *Let $\mathcal{S}$ be an IOLTS and $\mathcal{I}$ be an input-complete IOLTS with same visible alphabet ($\Sigma^?_{\mathcal{S}} = \Sigma^?_{\mathcal{I}}$ and $\Sigma^!_{\mathcal{S}} = \Sigma^!_{\mathcal{I}}$),*
*$\mathcal{I}$ **ioco** $\mathcal{S} \triangleq \forall \sigma \in \mathsf{STraces}(\mathcal{S}), Out(\Delta(\mathcal{I})\ \texttt{after}\ \sigma) \subseteq Out(\Delta(\mathcal{S})\ \texttt{after}\ \sigma)$.*

It can be proved [10] that $\mathcal{I}$ **ioco** $\mathcal{S} \iff \mathsf{STraces}(\mathcal{I}) \cap \mathsf{MinFTraces}(\mathcal{S}) = \emptyset$, where $\mathsf{MinFTraces}(\mathcal{S}) \triangleq \mathsf{STraces}(\mathcal{S}).\Sigma^!_{\mathcal{S}} \setminus \mathsf{STraces}(\mathcal{S})$ is the set of non-conformant suspension traces, minimal for the prefix ordering.

Test cases, test suites, properties: The behaviour of a test case is modelled by an IOLTS equipped with colours representing verdicts assigned to executions.

Definition 5 *A* test case *for $\mathcal{S}$ is a deterministic and input-complete IOLTS $\mathcal{TC} = (Q_{\mathcal{TC}}, \Sigma_{\mathcal{TC}}, \Lambda_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}}, \mathcal{C}_{\mathcal{TC}}, \texttt{init}_{\mathcal{TC}})$ where Pass, Fail, Inc, None $\in \Lambda_{\mathcal{TC}}$ are colours characterising verdicts. $\mathcal{C}_{\mathcal{TC}}$(Pass), $\mathcal{C}_{\mathcal{TC}}$(Fail), $\mathcal{C}_{\mathcal{TC}}$(Inc) and $\mathcal{C}_{\mathcal{TC}}$(None) forms a partition of $Q_{\mathcal{TC}}$. Its alphabet is $\Sigma_{\mathcal{TC}} = \Sigma^?_{\mathcal{TC}} \cup \Sigma^!_{\mathcal{TC}}$ where $\Sigma^?_{\mathcal{TC}} = \Sigma^{!\delta}_{\mathcal{S}}$ and $\Sigma^!_{\mathcal{TC}} = \Sigma^?_{\mathcal{S}}$ (outputs of $\mathcal{TC}$ are inputs of $\mathcal{S}$ and vice versa).* A test suite *is a set of test cases.*

The execution of a test case $\mathcal{TC}$ against an implementation $\mathcal{I}$ can be modelled by the parallel composition $\mathcal{TC} \| \mathcal{I}$ where common actions (inputs, outputs and quiescence)

[5] We here consider both loops or internal actions and divergences, *i.e.* infinite sequences of internal actions traversing an infinite number of states

are synchronized. The effect is to intersect sets of suspension traces ($\mathsf{Traces}(\mathcal{TC}\|\mathcal{I}) = \mathsf{STraces}(\Delta(\mathcal{I})) \cap \mathsf{Traces}(\mathcal{TC})$). Consequently, the possible failure of a test case on an implementation is defined as $\mathcal{TC}$ *fail* $\mathcal{I} \triangleq \mathsf{STraces}(\Delta(\mathcal{I})) \cap \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) = \emptyset$. Similar definitions can be given for *pass* and *inconc* relative to *Pass* and *Inc*.

We now define some properties that should be satisfied by test cases in order to correctly relate conformance to rejection by a test case:

Definition 6 *Let $\mathcal{S}$ be a specification, and $\mathcal{TS}$ a test suite for $\mathcal{S}$.*
TS is sound *if no test case may reject a conformant implementation:*
$\forall \mathcal{I}, \forall \mathcal{TC} \in \mathcal{TS}, \mathcal{I}$ **ioco** $\mathcal{S} \implies \neg(\mathcal{TC}$ *fail* $\mathcal{I})$.
$\mathcal{TS}$ is exhaustive *if it rejects all non-conformant implementations:*
$\forall \mathcal{I}, \neg(\mathcal{I}$ **ioco** $S) \implies \exists \mathcal{TC} \in \mathcal{TS}, \mathcal{TC}$ *fail* $\mathcal{I}$.
It is complete *if it is both sound and exhaustive.*
TS is strict *if it detects non-conformance as soon as they happen:*
$\forall \mathcal{I}, \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{TC}\|\mathcal{I}$ **ioco** $\mathcal{S}) \Rightarrow \mathcal{TC}$ *fail* $\mathcal{I}$.

The following characterisations derived from [10] are very convenient to prove those properties on generated test suites:

Proposition 1 *Let $\mathcal{TS}$ be a test suite for $\mathcal{S}$,*
$\mathcal{TS}$ is sound if $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \subseteq \mathsf{MinFTraces}(\mathcal{S}).\Sigma_{\mathcal{S}}^*$,
$\mathcal{TS}$ is exhaustive if $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \supseteq \mathsf{MinFTraces}(\mathcal{S})$,
$\mathcal{TS}$ is strict if $\bigwedge_{\mathcal{TC} \in \mathcal{TS}}(\mathsf{Traces}(\mathcal{TC}) \cap \mathsf{MinFTraces}(\mathcal{S}) \subseteq \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}))$.

3 Recursive Tiles Systems and their properties

In this section, we define the *Recursive Tiles Systems* (RTS), a model to define infinite state IOLTS based on the regular graphs of [7]. We present some key properties of these systems relative to ε-closure (suppression of internal actions), product and determinization that will be useful for test generation in the next sections.

Definition 7 *A* recursive tile system *(RTS) is a tuple $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ where*

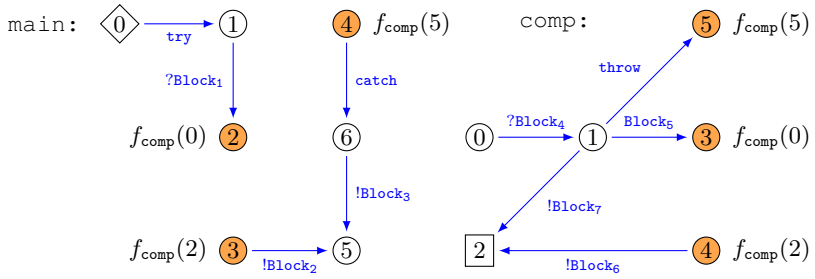
- $\Sigma = \Sigma_? \cup \Sigma_! \cup \Sigma_\tau$ *is a finite alphabet of* actions *partitioned into inputs, outputs and internal actions,*
- Λ *is a finite set of* colours *with a particular one* `init` *marking initial states.*
- $\mathcal{T}$ *is a set of tiles $t_\mathcal{A} = ((\Sigma, \Lambda), Q_\mathcal{A}, \rightarrow_\mathcal{A}, \mathcal{C}_\mathcal{A}, F_\mathcal{A})$ defined on (Σ, Λ) where*
  - $Q_\mathcal{A} \subseteq \mathbb{N}$ *is the set of* vertices*,*
  - $\rightarrow_\mathcal{A} \subseteq Q_\mathcal{A} \times \Sigma \times Q_\mathcal{A}$ *is a finite set of* transitions*,*
  - $\mathcal{C}_\mathcal{A} \subseteq Q_\mathcal{A} \times \Lambda$ *is a finite set of* coloured vertices*,*
  - $F_\mathcal{A} \subseteq \mathcal{T} \times 2^{\mathbb{N}\times\mathbb{N}}$*, the* frontier*, relates to some* tile*, $t_\mathcal{B}$, a partial function (often denoted $f_\mathcal{B}$) over $\mathbb{N}$, associating to vertices of $Q_\mathcal{B}$, vertices of $Q_\mathcal{A}$.*
- $t_0 \in \mathcal{T}$ *is an initial tile (the axiom).*

The frontier $F_\mathcal{A}$ of a tile $t_\mathcal{A}$ is used to append tiles $t_\mathcal{B}$ to $t_\mathcal{A}$: the frontier of $t_\mathcal{A}$ identifies tiles $t_\mathcal{B}$ and how some vertices of $t_\mathcal{B}$ are merged with vertices of $t_\mathcal{A}$.

A tile $t_\mathcal{A}$ defines an IOLTS $[t_\mathcal{A}] = (Q_\mathcal{A}, \Sigma, \Lambda, \rightarrow_\mathcal{A}, \mathcal{C}_\mathcal{A}, \texttt{init})$.

Example 1 *The following example presents an RTS abstracting the program of Fig. 1,* $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_{\texttt{main}})$ *with* $\Sigma^{\tau} = \{\texttt{try}, \texttt{throw}, \texttt{catch}, \texttt{Block}_5\}$, $\Sigma^{?} = \{?\texttt{Block}_1, ?\texttt{Block}_4\}$, $\Sigma^{!} = \{!\texttt{Block}_2, !\texttt{Block}_3, !\texttt{Block}_6, !\texttt{Block}_7\}$, $\Lambda = \{\texttt{init}, \texttt{succ}\}$, $\mathcal{T} = \{t_{\texttt{main}}, t_{\texttt{comp}}\}$ *a set of tiles, and* $t_{\texttt{main}}$ *the initial tile.*

- $t_{\texttt{main}} = ((\Sigma, \Lambda), Q_{\texttt{main}}, \rightarrow_{\texttt{main}}, \mathcal{C}_{\texttt{main}}, F_{\texttt{main}})$ *with*
  $Q_{\texttt{main}} = \{0, 1, 2, 3, 4, 5, 6\}$, $\mathcal{C}_{\texttt{main}} = \{(0, \texttt{init})\}$ *(*$\texttt{init}$ *depicted by* $\diamond$*)*
  $F_{\texttt{main}} = \{(\texttt{comp}, \{0 \rightarrow 2, 2 \rightarrow 3, 5 \rightarrow 4\})\}$, *and* $\rightarrow_{\texttt{main}}$ *depicted below,*
- $t_{\texttt{comp}} = (\texttt{comp}, (X, \Sigma, \Lambda), Q_{\texttt{comp}}, \rightarrow_{\texttt{comp}}, \mathcal{C}_{\texttt{comp}}, F_{\texttt{comp}})$ *with*
  $Q_{\texttt{comp}} = \{0, 1, 2, 3, 4, 5\}$, $\rightarrow_{\texttt{comp}}$ $\mathcal{C}_{\texttt{comp}} = \{(2, \texttt{succ})\}$ *(*$\texttt{succ}$ *depicted by* $\square$*),*
  $F_{\texttt{comp}} = \{(\texttt{comp}, \{0 \rightarrow 3, 2 \rightarrow 4, 5 \rightarrow 5\})\}$ *and* $\rightarrow_{\texttt{comp}}$ *depicted below.*



For the frontier, e.g., in the tile $t_{\texttt{main}}$, $f_{\texttt{comp}}(0)$ (2) *means that* $(\texttt{comp}, \{0 \rightarrow 2\})$ *belongs to* $F_{\texttt{main}}$*, i.e. the vertex* 0 *of* $t_{\texttt{comp}}$ *is associated to the vertex* 2 *of* $t_{\texttt{main}}$*.*

The semantics of an RTS is formally defined by an IOLTS by a *tiling* operation that appends tiles to another tile (initially, the axiom), inductively defining an IOLTS. Formally, given a set of tiles $\mathcal{T}$ and a tile $t_{\mathcal{E}} = ((\Sigma, \Lambda), Q_{\mathcal{E}}, \rightarrow_{\mathcal{E}}, \mathcal{C}_{\mathcal{E}}, F_{\mathcal{E}})$ with $F_{\mathcal{E}}$ defined on $\mathcal{T}$, the tiling of $t_{\mathcal{E}}$ by $\mathcal{T}$, denoted by $\mathcal{T}(t_{\mathcal{E}})$, is the tile $t'_{\mathcal{E}} = ((\Sigma, \Lambda), Q'_{\mathcal{E}}, \rightarrow'_{\mathcal{E}}, \mathcal{C}'_{\mathcal{E}}, F'_{\mathcal{E}})$ iteratively defined according to the elements of the frontier $F_{\mathcal{E}}$, as follows:

1. Initially, $Q'_{\mathcal{E}} = Q_{\mathcal{E}}$, $\rightarrow'_{\mathcal{E}} = \rightarrow_{\mathcal{E}}$, $\mathcal{C}'_{\mathcal{E}} = \mathcal{C}_{\mathcal{E}}$ $F'_{\mathcal{E}} = \emptyset$;
2. for each pair $(t_{\mathcal{B}}, f_{\mathcal{B}}) \in F_{\mathcal{E}}$, with $t_{\mathcal{B}} = ((\Sigma, \Lambda), Q_{\mathcal{B}}, \rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, F_{\mathcal{B}}) \in \mathcal{T}_{\mathcal{B}}$,
   let $\varphi_{\mathcal{B}} : Q_{\mathcal{B}} \rightarrow \mathbb{N}$ be the injection mapping vertices of $Q_{\mathcal{B}}$ to new vertices of $Q'_{\mathcal{E}}$ with $\varphi_{\mathcal{B}}(n) := f_{\mathcal{B}}(n)$ whenever $n \in dom(f_{\mathcal{B}})$, $n + \max(Q'_{\mathcal{E}}) + 1$ otherwise, where $\max(Q'_{\mathcal{E}})$ is the vertex with greatest value in $Q'_{\mathcal{E}}$. The tile $t'_{\mathcal{E}}$ is then defined by:
   - $Q'_{\mathcal{E}} = Q'_{\mathcal{E}} \cup Im(\varphi_{\mathcal{B}})$,
   - $\rightarrow'_{\mathcal{E}} = \rightarrow'_{\mathcal{E}} \cup \{(\varphi_{\mathcal{B}}(n), a, \varphi_{\mathcal{B}}(n')) \mid (n, a, n') \in \rightarrow_{\mathcal{B}}\}$,
   - $\mathcal{C}'_{\mathcal{E}} = \mathcal{C}'_{\mathcal{E}} \cup \{(\varphi_{\mathcal{B}}(n), \lambda) \mid (n, \lambda) \in \mathcal{C}_{\mathcal{B}}\}$,
   - $F'_{\mathcal{E}} = F'_{\mathcal{E}} \cup \{(t_C, \{(\varphi_{\mathcal{B}}(j), f_C(j)) \mid j \in dom(f_C)\}) \mid (t_C, f_C) \in F_{\mathcal{B}}\}$. The update of F' expresses that the frontier of the new tile $t'_{\mathcal{A}}$ is composed from those of the tiles that have been added.

Remark 1 *In a tiling, the order chosen to append a copy of the tiles that belong to the frontier is not important. Two different orders would produce isomorphic tiles (up to a renaming of vertices).*

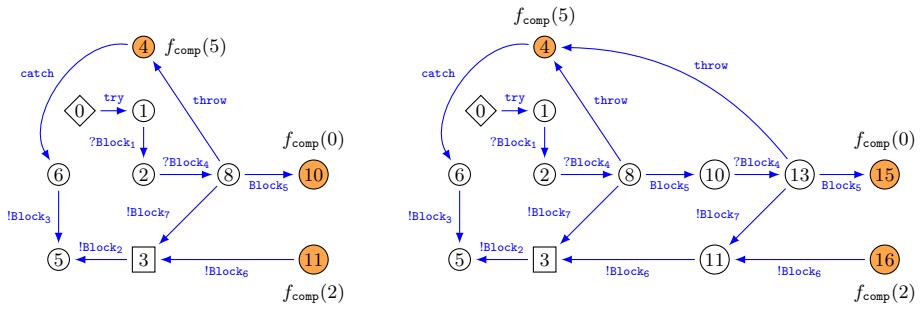


Fig. 2. $\mathcal{T}(t_{\mathtt{main}})$ and $\mathcal{T}^2(t_{\mathtt{main}})$ tiles

Example 2 *We illustrate the principle of tiling using the RTS defined in Example 1. Consider that $t_{\mathtt{main}}$ is the initial tile. Its tiling $\mathcal{T}(t_{\mathtt{main}})$, is performed as follows: there is a single element in its frontier; we add a copy of $t_{\mathtt{comp}}$ (with new vertices), identifying vertices $2, 3$ and 4 of $t_{\mathtt{main}}$ to vertices $0, 2$ and 5 of $t_{\mathtt{comp}}$.*

The resulting tile is depicted in Fig. 2 (left-hand side). This new tile may be in turn extended by adding a copy of $t_{\mathtt{comp}}$, identifying $4, 10$ and 11 to $0, 2$ and 5. Again, we illustrate the resulting tile in Fig. 2 (right-hand side) (observe that our definition of $\varphi_{\mathtt{comp}}$ induces that some elements of $\mathbb{N}$ are left out). Obviously iterating this process will result in vertex 4 having infinite in-*degree.*

An IOLTS is finally obtained from an RTS as the union of the IOLTS of tiles resulting from the iterated tilings from the axiom. Formally,

Definition 8 *Let $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ be an RTS. $\mathcal{R}$ defines an IOLTS $[\![\mathcal{R}]\!] = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \mathtt{init})$ given by*

$$\bigcup_k [\mathcal{T}^k(t_0)]$$

The infinite union of Definition 8 is valid because, by construction, for all $k \geq 0$: $[\mathcal{T}^k(t_0)] \subseteq [\mathcal{T}^{k+1}(t_0)]$, where $\subseteq$ is understood as the inclusion of IOLTS, *i.e.* inclusion of states, transitions and colourings.

For an RTS $\mathcal{R}$ with axiom t_0, and a state q in $[\![\mathcal{R}]\!]$, $\ell(q)$ denotes the *level* of q, *i.e.* the least $k \in \mathbb{N}$ such that q is a state of $[\mathcal{T}^k(t_0)]$, and $t(q)$ denotes the tile in $\mathcal{T}$ that created q. For a vertex v of a tile of $\mathcal{R}$, $[\![v]\!]$ denotes the set of states in $[\![\mathcal{R}]\!]$ corresponding to v.

Requirement 1 *In order to simplify proofs, we impose some technical restrictions on the RTS, $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, that can be ensured by a normalisation step, without loss of generality:*

1. *for any state, q, of finite degree in $[\![\mathcal{R}]\!]$, every transition connected to q is either defined in $t(q)$ or one of the tiles of its frontier (this may be checked on $\mathcal{T}$)*
2. *the set of enabled actions in copies of a vertex v is uniform (for all vertices v in $\mathcal{R}$, for all q, q' in $[\![v]\!]$, $\Gamma_{[\![\mathcal{R}]\!]}(q) = \Gamma_{[\![\mathcal{R}]\!]}(q')$), thus can be written $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!])$. Furthermore, we may assume that each vertex possesses a colour reflecting this value (see Corollary 1 below).*

Remark 2 *The IOLTS obtained from RTS correspond to the equational, or regular graphs of [7] and [3]. These IOLTS are derived from an axiom using deterministic* HR-

grammars. Each such grammar may be transformed into a tiling system, and conversely. Our definition aims at a greater simplicity.

Reachability Computation of (co)reachability sets, that are central for verification and safety problems, as well as for test generation, are effective for RTS:

Proposition 2 ([3]) *Given an RTS $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, a sub-alphabet $\Sigma' \subseteq \Sigma$, a colour $\lambda \in \Lambda$, and a new colour $r_\lambda \notin \Lambda$, an RTS $\mathcal{R}' = ((\Sigma, \Lambda \cup \{r_\lambda\}), \mathcal{T}', t'_0)$ can be effectively computed, such that $[\![\mathcal{R}']\!]$ is isomorphic to $[\![\mathcal{R}]\!]$ with respect to the transitions and the colouring by Λ, and states reachable from a state coloured λ by actions in Σ' are coloured r_λ: $\mathcal{C}_{\mathcal{R}'}(r_\lambda) = reach_{[\![\mathcal{R}']\!]}(\mathcal{C}(\lambda), \Sigma')$. The same result holds for states co-reachable from λ.*

Proposition 3.13 (b) of [3] enables to perform several computations related to our purpose. We rephrase it for RTS.

Proposition 3 ([3]) *Given an RTS $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, for any subset S in $\mathbb{N} \cup \{\infty\}$ and new colour $\#_S \notin \Lambda$, it is possible to compute an RTS $\mathcal{R}' = ((\Sigma, \Lambda \cup \{\#_S\}), \mathcal{T}', t'_0)$ such that $[\![\mathcal{R}]\!]$ is isomorphic to $[\![\mathcal{R}']\!]$ with respect to the transitions and the colouring by Λ, and every state of $[\![\mathcal{R}']\!]$ of (in- or out- or total-) degree is in S is coloured by $\#_S$.*

In particular this result enables to identify on the set of tiles properties of the states, like deadlocks, inputlock. The following corollary is also a direct consequence of this proposition (performing successive colouring for computing the degree related to some actions).

Corollary 1 *Given an RTS $\mathcal{R}$ and a vertex v of a tile t of $\mathcal{R}$, for any q in $[\![v]\!]$ the allowed actions $\Gamma_{[\![\mathcal{R}]\!]}(q)$ in state q can be effectively computed.*

Observable behaviour of RTS: Abstracting away internal transitions is important for test generation. With the following proposition, it is possible to do it for RTS.

Proposition 4 *From an RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!] = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \texttt{init})$ and visible actions $\Sigma^o \subseteq \Sigma$, one can effectively compute an RTS $Clo(\mathcal{R})$ with same colours Λ, whose IOLTS $[\![Clo(\mathcal{R})]\!] = (Q'_{\mathcal{R}}, \Sigma^o, \Lambda, \rightarrow'_{\mathcal{R}}, \mathcal{C}'_{\mathcal{R}}, \texttt{init})$ has no internal action, is of finite* out*-degree, and for any colour $\lambda \in \Lambda$, $\mathsf{Traces}_{\mathcal{C}_{\mathcal{R}}(\lambda)}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}'_{\mathcal{R}}(\lambda)}([\![Clo(\mathcal{R})]\!])$.*

This result is classical and follows mainly from [3]. Infinite out-degree may occur whenever there is an infinite sequence of internal transitions. However, careful computation of $Clo(\mathcal{R})$ enables to avoid such occurrences.

Synchronous product: The synchronous product of IOLTS is the operation used to intersect languages, and is useful for test selection using a test purpose. We can prove that the product of an RTS with a finite IOLTS is an RTS. More precisely, given any RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!]$, and a finite state IOLTS $\mathcal{A}$, one can compute an RTS denoted

by $\mathcal{R} \times \mathcal{A}$ such that $[\![\mathcal{R} \times \mathcal{A}]\!] = [\![\mathcal{R}]\!] \times \mathcal{A}$ (the $\times$ on the right-hand side of the equality is the product for IOLTS).

In general, the product of two RTS is not recursive. Indeed, the intersection of two context-free languages can be obtained by a product of two RTS, if such a product was recursive the intersection of two context-free languages would be a context-free language (*e.g.*, $\{a^n b^n c^k \mid n, k \in \mathbb{N}\} \cap \{a^n b^k c^k \mid n, k \in \mathbb{N}\}$ is not context-free).

Weighted RTS In the following we will often consider an important class of RTS. This class possesses the valuable property of being determinizable.

Definition 9 *An RTS $\mathcal{R}$ with IOLTS $[\![\mathcal{R}]\!] = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \texttt{init})$ is* weighted *if $\mathcal{C}_{\mathcal{R}}(\texttt{init})$ is a singleton $\{q_0\}$, and for any $u \in \Sigma^*$ and any states $q, q' \in Q_{\mathcal{R}}$, $q_0 \xrightarrow{u} q$ and $q_0 \xrightarrow{u} q'$ implies $\ell(q) = \ell(q')$ (same level).*

Note that determining if an RTS is weighted is decidable, using an algorithm from [5].

Example 3 *Assuming internal actions are not observable, the RTS defined in Example 1 may be weighted or not depending on the way the closure is performed. A backward closure ensures that the IOLTS is weighted: in fact, it is, then, deterministic. A forward closure induces non-determinism at $\texttt{?Block}_4$. Since path ending with this block would either be silently followed by $\texttt{throw}$ and thus end in the initial tile (level 0), or be followed by $\texttt{Block}_5$ and terminate at the next level (at least 1).*

Determinization of recursive LTS An RTS $\mathcal{R}$ is *deterministic* if its underlying IOLTS $[\![\mathcal{R}]\!]$ is deterministic. This is decidable from the set of tiles defining it (for example using Proposition 3). However, since PDA cannot be determinized in general, there is no hope to determinize an arbitrary RTS. Still, there are some classes of determinizable PDA, like visibly PDA [2], or, more recently, the *weighted grammars* of [4]. These grammars define a class of PDA that can be determinized and which both subsume the visibly PDA and the height deterministic PDA [11].

Proposition 5 ([5]) *Any weighted RTS $\mathcal{R}$ can be transformed into a* deterministic *one $\mathcal{D}(\mathcal{R})$ with same set of traces and, for any colour, same traces accepted in this colour.*

Example 4 *Following Example 3, assume that vertex 5 is not in any frontier anymore, and suppose that there are 3 transitions labelled $\texttt{?Block}_4$ between 0 and respectively 1, 3 and 5. This is a weighted system. In such a situation, determinization would simply perform a finite LTS determinization in the tile $\mathrm{t}_{\texttt{comp}}$. In the general case some tiles need to be merged first.*

4 Off-line test generation for weighted RTS

In this section and the following, we consider the generation of test cases from RTS. We focus, here, on weighted RTS, which are determinizable, and propose an off-line test generation algorithm that operates a selection guided by a test purpose (specified by a finite IOLTS). Computations are performed at the RTS level with consequences on the underlying IOLTS semantics, enabling the proof of properties on generated test cases.

4.1 Construction of the canonical tester

Quiescence As seen in Section 2 quiescence represents the absence of action in the specification. Given a specification defined by a RTS $\mathcal{S}$, detecting vertices where the absence of reaction is permitted enables to construct a suspended specification, $\Delta(\mathcal{S})$.

For finite state IOLTS, livelocks come from loops. On the contrary, for IOLTS defined by RTS, livelocks may come from infinite paths of silent actions involving infinitely many states. We call such paths *divergent*.

Lemma 1 *For a RTS $\mathcal{R}$, there exists a loop or a divergent path in $[\![\mathcal{R}]\!]$ if and only if there exists a vertex v and two states $q_1, q_2 \in [\![v]\!]$ with $\ell(q_1) \leq \ell(q_2)$ such that $q_1 \xrightarrow{\sigma} q_2$ for some $\sigma \in {\Sigma^\tau}^*$ and for all states q on this path, $\ell(q_1) \leq \ell(q)$.*

Proof. ($\Rightarrow$) Let $p = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 ...$ be an infinite path in $[\![\mathcal{R}]\!]$, with $\forall k \in \mathbb{N}, a_k \in \Sigma^\tau$. If p contains a loop, there exits one state of minimal level in this loop, let q_1 be this state. Now consider an elementary path. As each state is only seen once, we build a sequence of states q_{i_k} such that $\forall i_k \leq j, \ell(q_{i_k}) \leq \ell(q_j)$. As there are only a finite number of vertices, there is a least one v such two states of $[\![v]\!]$ appear in this path. Let these two states be q_1 and q_2.

($\Leftarrow$) If there exist a vertex v and two states $q_1, q_2 \in [\![v]\!]$ with $\ell(q_1) = \ell(q_2)$ such that $q_1 \xrightarrow{\sigma} q_2$ for $\sigma \in {\Sigma^\tau}^+$, and for all states q on this path, $\ell(q_1) \leq \ell(q)$, then $q_1 = q_2$, since any path from two distinct occurrences of the same tile at the same level involves vertices of lower level. Hence this path is a loop. Otherwise, $\ell(q_1) < \ell(q_2)$, let $p_0 := q_1 \xrightarrow{\sigma} q_2$ for $\sigma \in {\Sigma^\tau}^+$, since for all q in this path, $\ell(q_1) \leq \ell(q)$. Thus, by definition, a similar path may be constructed reaching a state q_3, with, $q_2 \xrightarrow{\sigma'} q_3$ for $\sigma' \in {\Sigma^\tau}^+$, $\ell(q_2) < \ell(q_3)$, and $\ell(q_2) \leq \ell(q)$ for all q involved. Iterating this process enables to produce an infinite path in $[\![\mathcal{R}]\!]$ satisfying the hypothesis. □

Proposition 6 *From any RTS $\mathcal{R}$, it is effective to build an RTS denoted $\Delta(\mathcal{R})$ such that $[\![\Delta(\mathcal{R})]\!] = \Delta([\![\mathcal{R}]\!])$. Consequently* $\mathsf{Traces}([\![\Delta(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$.

Proof. Let $\mathcal{R}$ be a RTS, we add self-loops δ as follows.
For deadlock and output lock, we use Requirement 1, item 2, which ensures that for a vertex v in a tile t of $\mathcal{R}$, has a uniform value for $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!])$. The δ-transitions are added to each v in $\mathcal{R}$ such that $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!]) = \emptyset$ or $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!]) \subseteq \Sigma^{\mathcal{R}}_?$. This operation produces a new RTS $\mathcal{R}'$.
For livelocks, there are two different cases: internal loops and *divergent paths*. From Lemma 1 we know that such situations may be detected from self-reaching vertices. This result also ensures that this detection may be performed taking each tile as an axiom. Then, for each tile t in $\mathcal{R}'$:

- Colour each vertex v of tile t by a colour λ_v not in $\Lambda_{\mathcal{R}'}$.
- Use Proposition 2 to colour by λ'_v vertices in $reach_{[\![\mathcal{R}_t]\!]}(\Sigma^\tau, \lambda)$, where $\mathcal{R}'_t$ is the RTS identical to $\mathcal{R}'$, with initial tile t. This computation simply enables to detect vertices involved in an infinite path, but the resulting RTS is not kept.
- Each vertex v coloured by both λ_v and λ'_v is involved in a livelock. We add quiescence to each such vertex in $\mathcal{R}'$ to produce $\Delta(\mathcal{R})$.

□

Output completion After using Proposition 6 for the computation of $\Delta(\mathcal{S})$ from the specification $\mathcal{S}$, the next step is to complete $\Delta(\mathcal{S})$ to recognise $\mathsf{STraces}(\mathcal{S}).\Sigma^{!\delta}$. The complete suspended specification, denoted by $CS(\mathcal{S})$, is computed from $\Delta(\mathcal{S})$ as follows: a new colour *UnS* is added to detect paths leading to unspecified behaviours. Then, for every tile t, a new vertex, v_t^{UnS}, is added (having colour *UnS*), new transitions leading to v_t^{UnS} are added as well:

$$\left\{ v \xrightarrow{a} v^{UnS} \mid v \in Q_A \wedge a \in \Sigma^{!\delta} \wedge a \notin \Gamma_{[\![\Delta(\mathcal{S})]\!]}([\![v]\!]) \right\}.$$

By construction, we get $\mathsf{Traces}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!]).\Sigma_{\mathcal{S}}^{!\delta} \cup \mathsf{STraces}([\![\mathcal{S}]\!])$ and $\mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$.

Canonical tester Whenever $CS(\mathcal{S})$ is weighted, Proposition 5 enables to determinize it into $\mathcal{D}(CS(\mathcal{S}))$. From $\mathcal{D}(CS(\mathcal{S}))$ we build a new RTS $Can(\mathcal{S})$ called the *canonical tester* of $\mathcal{S}$ as follows:

- a new colour *Fail* is considered and vertices of $\mathcal{D}(CS(\mathcal{S}))$ are coloured by *Fail* if composed of vertices all coloured by *UnS* in $CS(\mathcal{S})$.
- inputs and outputs are mirrored in $Can(\mathcal{S})$ wrt. $\mathcal{S}$.

From this construction we can deduce that

$$\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{MinFTraces}([\![\mathcal{S}]\!]) \quad (1)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!]) \quad (2)$$

and $\mathsf{Traces}([\![Can(\mathcal{S})]\!])$ is their disjoint union.

In fact $\mathsf{Traces}_{\overline{\mathcal{C}}(_{Fail})}([\![Can(\mathcal{S})]\!]) = \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) = \mathsf{STraces}([\![\mathcal{S}]\!])$ and

$$\begin{aligned}
\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) &= \mathsf{Traces}_{\mathcal{C}(_{UnS})}([\![CS(\mathcal{S})]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) \\
&= \mathsf{Traces}([\![CS(\mathcal{S})]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!]) \\
(\text{as}\mathsf{Traces}([\![CS(\mathcal{S})]\!]) \text{ is } &\text{ the union } \mathsf{Traces}_{\mathcal{C}(_{UnS})}([\![CS(\mathcal{S})]\!]) \cup \mathsf{Traces}_{\overline{\mathcal{C}}(_{UnS})}([\![CS(\mathcal{S})]\!])) \\
&= \mathsf{STraces}([\![\mathcal{S}]\!]).\Sigma_{\mathcal{S}}^{!\delta} \setminus \mathsf{STraces}([\![\mathcal{S}]\!]) \\
&= \mathsf{MinFTraces}([\![\mathcal{S}]\!])
\end{aligned}$$

From (1) it immediately follows that the test suite $\mathcal{TS}$ reduced to $\{Can(\mathcal{S})\}$ is sound and exhaustive (see Section 2). $\mathcal{TS}$ is also strict, which is proved as follows: $\mathsf{Traces}([\![Can(\mathcal{S})]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{S}]\!]) = (\mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!]) \cup \mathsf{STraces}([\![\mathcal{S}]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{S}]\!]) = \mathsf{Traces}_{\mathcal{C}(_{Fail})}([\![Can(\mathcal{S})]\!])$ using the disjoint union and (1).

Test case selection with a test purpose The canonical tester has important properties, but one may want to focus on particular behaviours, using a test purpose. In our formal framework, a *test purpose* is a deterministic finite IOLTS $\mathcal{TP}$ over $\Sigma^{o\delta}$, with a particular colour *Accept*. States coloured by *Accept* have no successors.

As seen in the previous section, the product $\mathcal{P}$ between $Can(\mathcal{S})$ and $\mathcal{TP}$ is an RTS. On this product, new colours are specified as follows :

- $\mathcal{C}_{\mathcal{P}}(\mathit{Fail}) = \mathcal{C}_{Can(\mathcal{S})}(\mathit{Fail}) \times Q_{\mathcal{TP}}$

- $\mathcal{C}_{\mathcal{P}}(Pass) = \overline{\mathcal{C}}_{\mathcal{P}}(Fail) \times \mathcal{C}_{\mathcal{TP}}(Accept)$
- $\mathcal{C}_{\mathcal{P}}(None) = Coreach(\mathcal{C}_{\mathcal{P}}(Pass)) \setminus \mathcal{C}_{\mathcal{P}}(Pass)$
- $\mathcal{C}_{\mathcal{P}}(Inc) = Q_P \setminus (\mathcal{C}_{\mathcal{P}}(Fail) \cup \mathcal{C}_{\mathcal{P}}(Pass) \cup \mathcal{C}_{\mathcal{P}}(None))$

Note that, by construction, each state has a unique colour in $\{Fail, Pass, None, Inc\}$. States coloured by *Fail* or *Pass* have no successors, and states coloured by *Inc* have only *Fail* or *Inc* successors.

In order to avoid states coloured by *Inc* where the test purpose cannot be satisfied anymore, transitions labelled by an output (input of $\mathcal{S}$, controllable by the environment) and leading to a state coloured by *Inc* may be pruned, as well as those leaving *Inc*. Consequently, runs leading to an *Inc* coloured state necessarily end with an input action.

Finally, the test case $\mathcal{TC}$ generated from $\mathcal{S}$ and $\mathcal{TP}$ is the product $\mathcal{P}$, equipped with new colours *Fail*, *Pass*, *None*, *Inc* and pruned as above.

Example 5 *Figure 3, below, represents the test case obtained from Example 1 with the test purpose accepting only* $(\Sigma^{\circ\delta})^{*}\,?\texttt{Block}_4\,?\texttt{Block}_4\,(\Sigma^{\circ\delta})^{*}\,!\texttt{Block}_2$. *The vertices labelled by* F *correspond to the one coloured by Fail but is split for better readability. Triangle vertices are those coloured by Inc. Observe that each vertex is a set of pairs, so indices depicted below are not related to the original ones.*

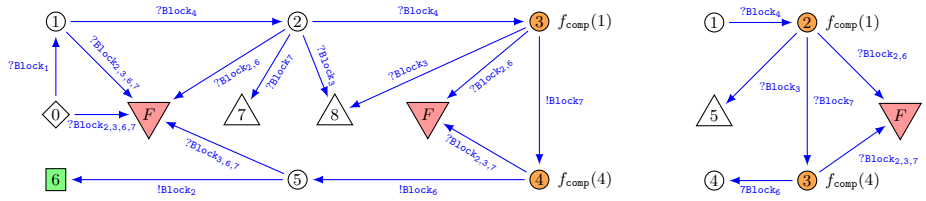


Fig. 3. Example of a test case

4.2 Properties of generated test cases

We now prove the requested properties of test cases defined in Section 2, relating test case failure to non-conformance, and a new property, precision, that relates test case success (*Pass* verdict) to the satisfaction of the test purpose.

Soundness and strictness According to the construction of $\mathcal{P}$, the definition of $\mathcal{C}_{\mathcal{P}}(Fail)$, and pruning, selection by $\mathcal{TP}$ do not add any colouring by *Fail* with respect to $Can(\mathcal{S})$, thus $\mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket) \cap \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket Can(\mathcal{S})\rrbracket)$. By (1) we deduce $\mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket) \cap \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket) \subseteq \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket)$ which proves both strictness (equality) and soundness (inclusion).

Exhaustiveness We prove that the test suite $\mathcal{TS}$ composed of all test cases that can be generated from arbitrary test purposes $\mathcal{TP}$ is exhaustive. We thus need to establish the inequality $\bigcup_{\mathcal{TC}\in\mathcal{TS}} \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket\mathcal{TC}\rrbracket) \supseteq \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket)$.

Let $\sigma' = \sigma.a \in \mathsf{MinFTraces}(\llbracket\mathcal{S}\rrbracket) = \mathsf{Traces}_{\mathcal{C}(Fail)}(\llbracket Can(S)\rrbracket)$ be a minimal non-conformant trace for $\mathcal{S}$. We have $\sigma \in \mathsf{STraces}(\llbracket\mathcal{S}\rrbracket)$ and there exists $b \in \Sigma^{!\delta}$ such

that $\sigma.b \in \mathsf{STraces}(\llbracket\mathcal{S}\rrbracket)$ (if no output continues σ in $\mathsf{STraces}(\llbracket\mathcal{S}\rrbracket)$, a δ does). Now consider a test purpose $\mathcal{TP}$ such that $\sigma.b \subseteq \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP})$ and let $\mathcal{TC}$ be the test case generated from $\mathcal{S}$ and $\mathcal{TP}$. By construction of $\mathcal{TC}$, we get $\sigma' \in \mathsf{Traces}_{Fail}(\llbracket\mathcal{TC}\rrbracket)$.

Precision As a complement to the above properties, *precision* relates test cases to test purposes. It says that the verdict *Pass* is returned as soon as possible, once the test purpose is satisfied. Formally, a test case $\mathcal{TC}$ is precise with respect to $\mathcal{TP}$ if $\mathsf{Traces}_{\mathcal{C}(Pass)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}(\llbracket\mathcal{S}\rrbracket) \cap \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket)$.

By construction, states coloured by *Pass* are those coloured by *Accept* in $\mathcal{TP}$ and not by *Fail* in $Can(\mathcal{S})$. Thus $\mathsf{Traces}_{\mathcal{C}(Pass)}(\llbracket\mathcal{TC}\rrbracket) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}(\llbracket\mathcal{S}\rrbracket)$ which (since $\mathsf{Traces}_{\mathcal{C}(Pass)}(\llbracket\mathcal{TC}\rrbracket) \subseteq \mathsf{Traces}(\llbracket\mathcal{TC}\rrbracket)$) implies precision.

5 On-line test generation from RTS

For the general case, determinization is an issue, as seen in Section 3. As usual in similar cases [13], one may rely on "on-line" test generation (executing test cases while generating them) or equivalently produce test cases as finite trees.

5.1 Test case generation

Output-completion and ϵ-closure The process starts from the output-completed specification $CS(\mathcal{S})$ defined in Section 4. This time, the canonical tester cannot be built from $CS(\mathcal{S})$. However, using Proposition 4, one can built $Clo(CS(\mathcal{S}))$, ensuring the following properties:

$$\mathsf{MinFTraces}(\mathcal{S}) \subseteq \mathsf{Traces}_{\mathcal{C}(UnS)}(Clo(CS(\mathcal{S}))) \subseteq \mathsf{STraces}(\mathcal{S}).\Sigma^{!\delta}$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}(Clo(CS(\mathcal{S}))) = \mathsf{STraces}(\mathcal{S})$$

Product and colouring The next step consists in the computation of the product of $Clo(CS(\mathcal{S}))$ with a test purpose given as a complete finite IOLTS $\mathcal{TP}$. Let $\mathcal{P} = Clo(CS(\mathcal{S})) \times \mathcal{TP}$ be this product, one may define the following new colours on $\mathcal{P}$ using a co-reachability analysis:

- $\mathcal{C}_{\mathcal{P}}(\mathit{UnS}) = \mathcal{C}_{Clo(CS(\mathcal{S}))}(\mathit{UnS}) \times Q_{\mathcal{TP}}$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Pass}) = \overline{\mathcal{C}}_{Clo(CS(\mathcal{S}))}(\mathit{UnS}) \times \mathcal{C}_{\mathcal{TP}}(\mathit{Accept})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{None}) = Coreach(\mathcal{C}_{\mathcal{P}}(\mathit{Pass})) \setminus \mathcal{C}_{\mathcal{P}}(\mathit{Pass})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Inc}) = Q_{\mathcal{P}} \setminus (\mathcal{C}_{\mathcal{P}}(\mathit{Fail}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{Pass}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{None}))$

Computing test cases The last step consists in computing test cases in a way similar to [13]. These test cases will be modelled as finite trees. Formally such a finite tree will be a prefix-closed set of words in $\Sigma^{o\delta^*}.(\{\mathit{Fail}, \mathit{Pass}, \mathit{None}, \mathit{Inc}\} \cup \{\varepsilon\})$. Given a tree θ, for some symbol a, the notation $a;\theta \triangleq \{au \mid u \in \theta\}$, furthermore, given two trees θ, θ', the tree formed by the union of those trees is denoted by $\theta + \theta'$.

A test case $\mathcal{TC}$ is a tree built from $\mathcal{P}$ by taking as argument a set of states PS. Let us define test cases by applying the following algorithm recursively, starting from the initial state $\mathcal{C}_{\mathcal{P}}(init)$.
Choose non deterministically between one of the following operations.

1. (* Terminate the test case *)
   $\theta := \{\mathit{None}\}$
2. (* Give a next input to the implementation *)
   Choose any $a \in out(PS)$ such that
   $(PS \texttt{ after } a) \cap (\mathcal{C}_{\mathcal{P}}(\mathit{Pass}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{None})) \neq \emptyset$
   $\theta := a; \theta'$
   where θ' is obtained by applying the algorithm with $PS' = (PS \texttt{ after } a)$
3. (* Check the next output of the implementation *)

$$\theta := \sum_{a \in X_1} a; \mathit{Fail} + \sum_{a \in X_2} a; \mathit{Inc} + \sum_{a \in X_3} a; \mathit{Pass} + \sum_{a \in X_4} a; \theta'$$

with:
- $X_1 = \{a \mid PS \texttt{ after } a \subseteq \mathcal{C}_{\mathcal{P}}(\mathit{UnS})\}$
- $X_2 = \{a \mid (PS \texttt{ after } a \subseteq (\mathcal{C}_{\mathcal{P}}(\mathit{Inc}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{UnS}))) \wedge (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(\mathit{Inc}) \neq \emptyset)\}$
- $X_3 = \{a \mid PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(\mathit{Pass}) \neq \emptyset\}$
- $X_4 = \{a \mid (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(\mathit{Pass}) = \emptyset) \wedge (PS \texttt{ after } a \cap \mathcal{C}_{\mathcal{P}}(\mathit{None}) \neq \emptyset)\}$
- θ' is obtained by applying the algorithm with $PS' = (PS \texttt{ after } a)$

Formally, a tree needs to be transformed into a test case IOLTS $\mathcal{TC}$ by an appropriate colouring of states ending in *Fail*, *Pass*, *Inc* or *None* after a suspension trace. We skip this for readability.

5.2 Properties of the test cases generated on-line

Soundness and Strictness By definition of X_1, those traces of $\mathcal{TC}$ falling in a state coloured by *Fail* are those in $\mathsf{Traces}(\llbracket CS(\mathcal{S}) \rrbracket) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}_{(UnS)}}(\llbracket CS(\mathcal{S}) \rrbracket) = \mathsf{MinFTraces}(\llbracket \mathcal{S} \rrbracket)$. Thus $\mathsf{Traces}_{\mathcal{C}(\mathit{Fail})}(\mathcal{TC}) = \mathsf{MinFTraces}(\llbracket \mathcal{S} \rrbracket) \cap \mathsf{Traces}(\mathcal{TC})$ which proves both soundness and strictness, as in the off-line case.

Exhaustiveness The proof of exhaustiveness is similar to the one in Section 4, consisting in building a test purpose $\mathcal{TP}$ for each non-conformant trace, and proving that a possible resulting test case would produce a *Fail* after this trace.

Precision From the construction of $\mathcal{TC}$, in particular, the set X_3, we have $\mathsf{Traces}_{\mathcal{C}(\mathit{Pass})}(\mathcal{TC}) = \mathsf{Traces}_{\mathcal{C}(\mathit{Pass})}(Clo(CS(\mathcal{S})) \times \mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$. Then, by definitions of the colours, we obtain: $\mathsf{Traces}_{\mathcal{C}(\mathit{Pass})}(\mathcal{TC}) = \mathsf{Traces}_{\overline{\mathcal{C}}_{UnS}}(Clo(CS(\mathcal{S}))) \cap \mathsf{Traces}_{\mathcal{C}(\mathit{Accept})}(\mathcal{TP})) \cap \mathsf{Traces}(\mathcal{TC})$. Which eventually proves precision: $\mathsf{Traces}_{\mathcal{C}(\mathit{Pass})}(\mathcal{TC}) = \mathsf{STraces}(\mathcal{S}) \cap \mathsf{Traces}_{\mathcal{C}(\mathit{Accept})}(\mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$.

6 Conclusion

In this paper we have presented recursive tile systems, a general model of IOLTS allowing for recursion. We have provided algorithms to produce sound, strict and exhaustive test suites, either off-line or on-line. These algorithms enable to employ test purposes (even, for the on-line case) which are a classical way to drive tests towards sensitive properties. We have also established the precision of our tests with respect to test purposes.

An interesting perspective would be to incorporate known results on probabilistic RTS. This would enable to take into account quantitative properties of systems, or to express coverage properties of finite test suites.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *13th International Conference on Computer Aided Verification, (CAV'01)*, volume 2102 of *LNCS*, pages 207–220, 2001.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*, pages 202–211. ACM, 2004.
3. D. Caucal. Deterministic graph grammars. In *Texts in logics and games 2*, pages 169–250, 2007.
4. D. Caucal. Synchronization of regular automata. In *34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, volume 5734 of *LNCS*, pages 2–23, 2009.
5. D. Caucal and S. Hassen. Synchronization of grammars. In *Third International Computer Science Symposium in Russia (CSR'08)*, volume 5010 of *LNCS*, pages 110–121, 2008.
6. C. Constant, B. Jeannet, and T. Jéron. Automatic test generation from interprocedural specifications. In *TestCom/FATES'07*, volume 4581 of *LNCS*, pages 41–57, 2007.
7. B. Courcelle. *Handbook of Theoretical Computer Science*, chapter Graph rewriting: an algebraic and logic approach. Elsevier, 1990.
8. L. Frantzen, J. Tretmans, and T. A. C. Willemse. A symbolic framework for model-based testing. In *FATES 2006 and RV 2006, Revised Selected Papers*, volume 4262 of *LNCS*, pages 40–54, 2006.
9. C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.
10. B. Jeannet, T. Jéron, and V. Rusu. Model-based test selection for infinite-state reactive systems. In *5th International Symposium on Formal Methods for Components and Objects (FMCO'06), Revised Lectures*, volume 4709 of *LNCS*, pages 47–69, 2006.
11. D. Nowotka and J. Srba. Height-deterministic pushdown automata. In *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*, volume 4708 of *LNCS*, pages 125–134, 2007.
12. G.J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.
13. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.