Test Generation from Recursive Tile Systems

Sébastien Chédor, Thierry Jéron, Christophe Morvan

Test Generation from Recursive Tile Systems

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

[1]Université de Rennes I, Campus de Beaulieu, 35042 Rennes, France , E-mail: sebastien.chedor@inria.fr
[2]INRIA Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes, France , E-mail: thierry.jeron@inria.fr
[3]Université Paris-Est, UPEMLV, F-77454, Marne-La-Vallée, France , E-mail: christophe.morvan@univ-paris-est.fr

Abstract

This paper explores the generation of conformance test cases for *Recursive Tile Systems* (RTSs) 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. Test generation for this kind of infinite state labelled transition systems is seldom explored in the literature. The first part presents an off-line test generation algorithm for *Weighted* RTSs, a determinizable sub-class of RTSs, and the second one, an on-line test generation algorithm for the full RTS model. Both algorithms use test purposes to guide test selection through targeted behaviours. Additionally, essential properties relating verdicts produced by generated test cases with both the soundness with respect to the specification, and the precision with respect to a test purpose, are proved.

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. Automation 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) using mathematical models, formal definitions of conformance and 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 1996 by Tretmans [1] is now a well established framework for the formal modelling of conformance testing for Input/Output Labelled Transition Systems (IOLTSs). Test generation algorithms and tools have been designed for this model [2, 3] and for more general models whose semantics can be expressed in the form of infinite state IOLTSs [4, 5]. Test generation techniques have also been devised for timed automata models whose semantics are infinite state systems [6, 7, 8].

One can distinguish two different approaches in test generation: *off-line* test generation aims at generating test cases, storing them, and later executing them on the implementation, while in *on-line* test generation, test cases are generated while executing them on the implementation, taking into account its reactions to stimuli of the test cases. In both cases, formal properties of test suites need be considered, for example, soundness reflects that no conformant implementation may be rejected, while exhaustiveness expresses that every non-conformant implementation is detected by at least one test in the suite.

When considering infinite state systems, undecidability is often an issue. Very simple models like two counters machines lead to the undecidability of the most basic properties (e.g., reachability of a given configuration, occurrence

of a given output). Furthermore, provided the description of a reactive system in a given model, the observable behaviour of such a system may not be expressible in this model. In order to establish properties like soundness and exhaustiveness of a generated test-suite, it is convenient to have both a formal description of the system and to be able to prove properties relative to the generated tests. There are several models between finite state and Turing powerful systems; this paper considers a variant of Pushdown automata (PDAs), which provide a nice middle-ground between expressivity and decidability. They form a model for reactive recursive programs, like the running example which represents an abstraction of the one in Figure 1. This example is presented in some Java-like syntax and involves exceptions (a shortcut that is used whenever the keyword `throw` is used). More precisely the program asks for some integer, then calls the recursive function `comp`, which asks for a boolean, and, depending on its value, proceeds by making a recursive call or stopping. Whenever exceptions are raised the program branches directly to the `catch` block of the `main` function. Along the paper we will only focus on the control flow of this program and abstract data values away.

```
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());
   }
}
int 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;
   }
}
```

Figure 1: A recursive program

There exist several ways to define recursive behaviours: PDAs, recursive state machines by Alur *et al.* [9] or regulars graphs, defined by functional (or deterministic) hyperedge replacement grammars (HR-grammars) [10, 11]. Each of these models has its merits and flaws: PDAs 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 and also enables to model systems having states of infinite degree. Furthermore, recent results by Caucal and Hassen define classes of such systems which may be determinized [12], which is of interest for test generation. The HR-grammars, on the other hand, are very technical to define. The present paper tries to get the best of both

worlds: HR-grammars are presented as tiling systems, called Recursive Tile Systems (RTSs for short). These systems have already been used in the context of diagnosis by Chédor *et al.* [13]; they are mostly finite sets of finite LTSs with frontiers, crossing the frontier corresponds to entering a new copy of one of the finite LTSs. Additionally, the alphabet of actions is partitioned into inputs, outputs and internal actions. The semantics of an RTS is then defined as an infinite state IOLTS. 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* RTSs, which permits to design off-line test generation algorithms for this sub-class. For the whole class of RTSs however, determinization is impossible, but on-line test generation is still possible as subset construction is performed along finite executions.

To the best knowledge of the authors, test generation for recursive programs has been seldom considered in the literature. The work of Constant *et al.* [14], which considers a model of deterministic PDA with inputs/outputs (IOPDS) and generates test cases in the same model is apparently the only previous work with PDAs. 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. Section 3 defines the model of RTS for the description of recursive reactive programs, gives its semantics in terms of an infinite state IOLTS obtained by recursive expansion of tiles. Section 4, in the **ioco** framework, proposes an off-line test selection algorithm guided by test purposes for *Weighted* RTSs, a determinizable sub-class of RTSs, and proves essential properties of generated test cases. Furthermore, Section 5 provides an on-line test generation algorithm for the full RTS model, also using test purposes for test selection. Eventually, properties of generated test cases are proved.

2 Conformance testing theory for IOLTSs

This section recalls the **ioco** testing theory introduced by Tretmans [1] for the model of Input/Output Labelled Transition Systems (IOLTSs) that will serve as a basis for test generation from RTSs. First a non-standard definition of IOLTSs is given, where marking of states is defined by colours, then notations and basic operations on IOLTSs are introduced. Afterwards several notions are reviewed: the **ioco** testing theory, with the modelling of test artefacts and their interactions, the central notion of conformance relation, and essential properties requested on test cases.

2.1 The IOLTS model and operations

There is a lot of literature relative to IOLTSs. This notion classically boils down to Kripke structures with an alphabet partitioned into inputs, outputs and internal actions. The following definition is not the classical one in the sense that it uses colours to identify sets of states. Furthermore, a colour is also used to single out initial states. Obviously this modification does not affect the properties of IOLTSs, however it will be useful later on in this paper when considering such systems defined by recursive tile systems.

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

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

Formula $q \xrightarrow[\mathcal{M}]{\mu} q'$ denotes $(q, \mu, q') \in \rightarrow_{\mathcal{M}}$ and $q \xrightarrow[\mathcal{M}]{\mu}$ denotes $\exists q' : q \xrightarrow[\mathcal{M}]{\mu} q'$. The first notation is generalized to sequences of actions, and thus $q \xrightarrow[\mathcal{M}]{w} q'$ denotes $\exists q_0, \ldots, q_n : q = q_0 \xrightarrow[\mathcal{M}]{\mu_1} q_1 \xrightarrow[\mathcal{M}]{\mu_2} \cdots \xrightarrow[\mathcal{M}]{\mu_n} q_n = q'$, with

[1] In the examples, for readability reasons, an input $a \in \Sigma^?$ is written $?a$, an output $x \in \Sigma^!$ is written $!x$. Internal actions have no extra symbol.

$w = \mu_1 \cdots \mu_n \in (\Sigma)^*$. Such an alternate sequence of states and labelled transitions is called a *path*. For each of these notations, the subscript $\mathcal{M}$ is omitted whenever there is no ambiguity with respect to the IOLTS.

The language of M *accepted* in a set of states $P \subseteq Q$ noted $L_P(\mathcal{M}) \triangleq \{w \in (\Sigma)^* \mid \exists q_0 \in \mathcal{C}(\texttt{init}), q \in P : q_0 \xrightarrow[\mathcal{M}]{w} q\}$, is the set of sequences from an initial state to a state in P. In particular $L(\mathcal{M}) \triangleq L_Q(\mathcal{M})$ represents the whole set of sequences of $\mathcal{M}$. A sequence is *accepted* in a colour λ if it is accepted in $\mathcal{C}(\lambda)$ and $L_\lambda(\mathcal{M})$ stands for $L_{\mathcal{C}(\lambda)}(\mathcal{M})$.

For $X \subseteq Q$ a subset of states and $\Sigma' \subseteq \Sigma$ a sub-alphabet, $post_{\mathcal{M}}(\Sigma', X) = \{q' \in Q \mid \exists q \in X, \exists \mu \in \Sigma' : q \xrightarrow[\mathcal{M}]{\mu} q'\}$ denotes the set of successors of a state in X by a single action in Σ', conversely $pre_{\mathcal{M}}(\Sigma', X) = \{q \in Q \mid \exists q' \in X, \exists \mu \in \Sigma' : q \xrightarrow[\mathcal{M}]{\mu} q'\}$ denotes the set of predecessors of X by a single action in Σ'. The set of states *reachable* from a set of states $P \subseteq Q$ 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 $P \subseteq Q$ (*i.e.* the set of states from which P is reachable) is $coreach_{\mathcal{M}}(\Sigma', P) \triangleq \text{lfp}(\lambda X.P \cup pre_{\mathcal{M}}(\Sigma', X))$. For a colour $\lambda \in \Lambda$, $reach_{\mathcal{M}}(\Sigma', \lambda)$ denotes $reach_{\mathcal{M}}(\Sigma', \mathcal{C}(\lambda))$ and $coreach_{\mathcal{M}}(\Sigma', \lambda)$ denotes $coreach_{\mathcal{M}}(\Sigma', \mathcal{C}(\lambda))$.

For a state q, $\Gamma_{\mathcal{M}}(q) \triangleq \{\mu \in \Sigma \mid q \xrightarrow[\mathcal{M}]{\mu}\}$ denotes the subset of actions enabled in q and respectively, $Out_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^!$ and $In_{\mathcal{M}}(q) \triangleq \Gamma_{\mathcal{M}}(q) \cap \Sigma^?$ denote the set of outputs (resp. inputs) enabled in q. The notation is generalized to sets of states: for $P \subseteq Q$, $Out_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} Out_{\mathcal{M}}(q)$ and $In_{\mathcal{M}}(P) \triangleq \bigcup_{q \in P} In_{\mathcal{M}}(q)$.

Visible behaviours of $\mathcal{M}$, which are essential to consider for testing, are defined as usual by the relation $\underset{\mathcal{M}}{\Rightarrow} \in Q \times (\{\epsilon\} \cup \Sigma^o) \times Q$ as follows: $q \underset{\mathcal{M}}{\overset{\varepsilon}{\Rightarrow}} q' \triangleq q = q'$ or $q \xrightarrow[\mathcal{M}]{\tau_1.\tau_2 \cdots \tau_n}{}^* q'$, for $\tau_i \in \Sigma^\tau$ and for $a \in \Sigma^o$, $q \underset{\mathcal{M}}{\overset{a}{\Rightarrow}} q' \triangleq \exists q_1, q_2 : q \underset{\mathcal{M}}{\overset{\varepsilon}{\Rightarrow}} q_1 \xrightarrow[\mathcal{M}]{a} q_2 \underset{\mathcal{M}}{\overset{\varepsilon}{\Rightarrow}} q'$. The notation is extended to sequences as follows: for $\sigma = a_1 \cdots a_n \in (\Sigma^o)^*$ a sequence of visible actions, $q \underset{\mathcal{M}}{\overset{\sigma}{\Rightarrow}} q'$ stands for $\exists q_0, \ldots, q_n : q = q_0 \underset{\mathcal{M}}{\overset{a_1}{\Rightarrow}} q_1 \cdots \underset{\mathcal{M}}{\overset{a_n}{\Rightarrow}} q_n = q'$ and $q \underset{\mathcal{M}}{\overset{\sigma}{\Rightarrow}}$ for $\exists\, q' : q \underset{\mathcal{M}}{\overset{\sigma}{\Rightarrow}} q'$.

The formula $q \ \texttt{after}\ \sigma \triangleq \{q' \in Q \mid q \underset{\mathcal{M}}{\overset{\sigma}{\Rightarrow}} q'\}$ denotes the set of states in which $\mathcal{M}$ can be after observing the visible sequence σ starting from the state q. The notation is extended to sets of states: for $P \subseteq Q$, $P \ \texttt{after}\ \sigma \triangleq \bigcup_{q \in P} q \ \texttt{after}\ \sigma$.

For a state q, $\mathsf{Traces}(q) \triangleq \{\sigma \in (\Sigma^o)^* \mid q \underset{\mathcal{M}}{\overset{\sigma}{\Rightarrow}}\}$ denotes the set of sequences of visible actions (called *traces*) that may be observed from q and $\mathsf{Traces}(\mathcal{M}) \triangleq \bigcup_{q_0 \in \mathcal{C}(\texttt{init})} \mathsf{Traces}(q_0)$ are those traces from the set of initial states. For a set of states P, $\mathsf{Traces}_P(\mathcal{M}) = \{\sigma \in (\Sigma^o)^* \mid (\mathcal{C}(\texttt{init}) \ \texttt{after}\, \sigma) \cap P \neq \emptyset\}$ 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. $\forall q \in Q, \forall \mu \in \Sigma^?, q \underset{\mathcal{M}}{\overset{\mu}{\Rightarrow}}$.

$\mathcal{M}$ is *complete in a state* q if any action is enabled in q: $\forall q \in Q, \Gamma(q) = \Sigma$. $\mathcal{M}$ is *complete* if it is complete in all states.

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

From an IOLTS $\mathcal{M}$, one can define a deterministic IOLTS $\mathcal{D}(\mathcal{M})$ with the same set of traces as $\mathcal{M}$ as follows: $\mathcal{D}(\mathcal{M}) = (2^Q, \Sigma^o, \Lambda_{\mathcal{D}}, \rightarrow_{\mathcal{D}}, \mathcal{C}_{\mathcal{D}}, \texttt{init}_{\mathcal{D}})$ where for $P, P' \in 2^Q$, $a \in \Sigma^o$, $P \xrightarrow[\mathcal{D}]{a} P' \iff P' = P \ \texttt{after}\, a$, and $init_{\mathcal{D}} \in \Lambda_{\mathcal{D}}$ is the colour for the singleton state $\mathcal{C}_{\mathcal{D}}(init_{\mathcal{D}}) = \mathcal{C}(\texttt{init}) \ \texttt{after}\, \varepsilon \in 2^Q$. 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 Λ and $\mathcal{C}$. For example, if $f \in \Lambda$ defines marked states in $\mathcal{M}$, one may define a colour $F \in \Lambda_{\mathcal{D}}$ for $\mathcal{D}(\mathcal{M})$ such that $\mathsf{Traces}_{\mathcal{C}(f)}(\mathcal{M}) = \mathsf{Traces}_{\mathcal{C}_{\mathcal{D}}(F)}(\mathcal{D}(M))$ simply by colouring by F the states $s \in 2^Q$ such that $\mathcal{C}(f)$ intersects s, *i.e.* at least one state in s is marked by f: $\mathcal{C}_{\mathcal{D}}(F) = \{s \in 2^Q \mid s \cap \mathcal{C}(f) \neq \emptyset\}$. Observe that the definition of $\mathcal{D}(\mathcal{M})$ is not always effective (meaning that the process may not always be carried out in finitely many steps by some algorithm). 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 IOLTSs: As usual, one may define a product of two IOLTSs such that sequences of actions in the product IOLTS are the sequences of actions of both IOLTSs. The product of IOLTSs thus implements the intersection of (accepted) languages:

Definition 2 (Synchronous product)**.** *Let* $\mathcal{M}_i = (Q_i, \Sigma, \Lambda_i, \rightarrow_i, \mathcal{C}_i, \texttt{init}_i)$*,* $i = 1, 2$ *be two IOLTSs with same alphabet* Σ*. Their synchronous product* $\mathcal{M}_1 \times M_2$ *is the IOLTS* $\mathcal{P} = (Q_\mathcal{P}, \Sigma, \Lambda_\mathcal{P}, \rightarrow_\mathcal{P}, \mathcal{C}_\mathcal{P}, \texttt{init}_\mathcal{P})$ *such that* $Q_\mathcal{P} \triangleq Q_1 \times Q_2$*, and* $\forall (q_1, q_2), (q_1', q_2') \in Q_\mathcal{P}$*,* $\forall \mu \in \Sigma$*,* $(q_1, q_2) \xrightarrow[\mathcal{P}]{\mu} (q_1', q_2') \triangleq q_1 \xrightarrow[\mathcal{M}_1]{\mu} q_1' \wedge q_2 \xrightarrow[\mathcal{M}_2]{\mu} q_2'$*. We define* $\Lambda_\mathcal{P} \triangleq \Lambda_1 \times \Lambda_2$*, in particular* $\texttt{init}_\mathcal{P} \triangleq (\texttt{init}_1, \texttt{init}_2)$*, and for any* $(\lambda_1, \lambda_2) \in \Lambda_\mathcal{P}$ *the colouring relation is defined by* $\mathcal{C}_\mathcal{P}((\lambda_1, \lambda_2)) \triangleq \mathcal{C}_1(\lambda_1) \times \mathcal{C}_2(\lambda_2)$*.*

As usual, for $P_i \subseteq Q_i, i = 1, 2$ the following holds: $L_{P_1 \times P_2}(\mathcal{M}_1 \times M_2) = L_{P_1}(M_1) \cap L_{P_2}(M_2)$ and in particular $L(\mathcal{M}_1 \times M_2) = L(M_1) \cap L(M_2)$ for the case where $P_i = Q_i, i = 1, 2$.

Parallel composition of IOLTSs: The parallel composition of IOLTSs is a binary operation used to formalize the synchronous interaction between two IOLTSs. In this interaction, inputs of one IOLTS are synchronized with outputs of the other one, and vice versa. This operation is used to describe the execution of test cases on an implementation.

Definition 3 (Parallel composition)**.** *Let* $\mathcal{M}_i = (Q_i, \Sigma_i, \Lambda_i, \rightarrow_i, \mathcal{C}_i, \texttt{init}_i)$*,* $i = 1, 2$ *be two IOLTSs with mirrored visible alphabets (*i.e. $\Sigma_1^! = \Sigma_2^?$ *and* $\Sigma_1^? = \Sigma_2^!$*). Their parallel composition is the IOLTS* $\mathcal{M}_1 \| \mathcal{M}_2 = (Q_\mathcal{M}, \Sigma_\mathcal{M}, \Lambda_\mathcal{M}, \rightarrow_\mathcal{M}, \mathcal{C}_\mathcal{M}, \texttt{init}_\mathcal{M})$ *with* $Q_\mathcal{M} = Q_1 \times Q_2$*,* $\Sigma_\mathcal{M} = \Sigma_1$*,* $\Lambda \triangleq \Lambda_1 \times \Lambda_2$*, in particular* $\texttt{init} \triangleq (\texttt{init}_1, \texttt{init}_2)$*, for any* $(\lambda_1, \lambda_2) \in \Lambda$ *the colouring relation is defined by* $\mathcal{C}((\lambda_1, \lambda_2)) \triangleq \mathcal{C}_1(\lambda_1) \times \mathcal{C}_2(\lambda_2)$*, and the transition relation is defined by the rules:*

$$\frac{a \in \Sigma^o \quad q_1 \xrightarrow[\mathcal{M}_1]{a} q_1' \quad q_2 \xrightarrow[\mathcal{M}_2]{a} q_2'}{(q_1, q_2) \xrightarrow[\mathcal{M}]{a} (q_1', q_2')} \qquad \frac{q_1 \xrightarrow[\mathcal{M}_1]{\tau} q_1'}{(q_1, q_2) \xrightarrow[\mathcal{M}]{\tau} (q_1', q_2)} \qquad \frac{q_2 \xrightarrow[\mathcal{M}_2]{\tau} q_2'}{(q_1, q_2) \xrightarrow[\mathcal{M}]{\tau} (q_1, q_2')}$$

Synchronization being defined on visible actions, thus, for $P_i \subseteq Q_i, i = 1, 2$ the following holds $\mathsf{Traces}_{P_1 \times P_2}(\mathcal{M}) = \mathsf{Traces}_{P_1}(\mathcal{M}_1) \cap \mathsf{Traces}_{P_2}(\mathcal{M}_2)$, and in particular $\mathsf{Traces}(\mathcal{M}) = \mathsf{Traces}_{P_1}(\mathcal{M}_2) \cap \mathsf{Traces}_{P_2}(\mathcal{M}_2)$. Note that this definition is not completely symmetric: the direction of actions (output, input) is given by the first operand.

2.2 The ioco testing theory

Specification and implementation: In the **ioco** testing framework, it is assumed that the behaviour of the specification is modelled by an IOLTS $\mathcal{M} = (Q, \Sigma, \Lambda, \rightarrow_\mathcal{M}, \mathcal{C}, \texttt{init})$. 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 $\Sigma_\mathcal{I} = \Sigma_\mathcal{I}^? \cup \Sigma_\mathcal{I}^! \cup \Sigma_\mathcal{I}^\tau$ and $\Sigma_\mathcal{I}^? = \Sigma^?$ and $\Sigma_\mathcal{I}^! = \Sigma^!$. The input-completeness assumption means that the implementation is always ready to receive inputs from its environment, in particular from test cases. In the sequel, the set of implementations, with alphabet compatible with $\mathcal{M}$, is denoted by $\mathcal{IMP}(\mathcal{M})$.

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.* $\Gamma(q) \subseteq \Sigma^?$, (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.* $\forall n \in \mathbb{N}, \exists \sigma \in (\Sigma^\tau)^n, q \xrightarrow[\mathcal{M}]{\sigma}$ [2]. Whenever q is an output quiescence or in a livelock is denoted by $quiescent(q)$. From an IOLTS $\mathcal{M}$ one can define a new IOLTS $\Delta(\mathcal{M})$ where quiescence is made explicit by a new output δ:

[2] While the original **ioco** theory restricts to non-divergent IOLTSs, in this paper, IOLTSs having both loops of internal actions and divergences, *i.e.* infinite sequences of internal actions traversing an infinite number of states, are considered.

Definition 4 (Suspension)**.** *Let* $\mathcal{M} = (Q, \Sigma, \Lambda, \rightarrow_{\mathcal{M}}, \mathcal{C}, \texttt{init})$ *be an IOLTS, the* suspension of $\mathcal{M}$ *is the IOLTS* $\Delta(\mathcal{M}) = (Q, \Sigma_{\Delta(\mathcal{M})}, \Lambda, \rightarrow_{\Delta(\mathcal{M})}, \mathcal{C}, \texttt{init})$ *where* $\Sigma_{\Delta(\mathcal{M})} = \Sigma \cup \{\delta\}$ *with* $\delta \in \Sigma^{!}_{\Delta(\mathcal{M})}$ *(δ is considered as an output, observable by the environment), and the transition relation* $\rightarrow_{\Delta(\mathcal{M})} \triangleq \rightarrow_{\mathcal{M}} \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$ *is obtained from* $\rightarrow_{M}$ *by adding δ loops for each quiescent state q.*

Note that $\Delta(\mathcal{M})$ might be not computable for infinite state IOLTSs. In the sequel, the sets $\Sigma^{!} \cup \{\delta\}$ and $\Sigma^{o} \cup \{\delta\}$ are respectively denoted by $\Sigma^{!\delta}$ and $\Sigma^{o\delta}$. The traces of $\Delta(\mathcal{M})$ denoted by $\mathsf{STraces}(\mathcal{M})$ are called the *suspension traces* of $\mathcal{M}$. They represent the visible behaviours 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 [1], given a specification IOLTS $\mathcal{M}$, an implementation $\mathcal{I} \in \mathcal{IMP}(\mathcal{M})$ is said to conform to its specification $\mathcal{M}$ if, after any suspension trace σ of $\mathcal{M}$, the implementation $\mathcal{I}$ exhibits only outputs and quiescences that are specified in $\mathcal{M}$. Formally:

Definition 5 (Conformance relation)**.** *Let $\mathcal{M}$ be an IOLTS and $\mathcal{I} \in \mathcal{IMP}(\mathcal{M})$ be an input-complete IOLTS with same visible alphabet (*i.e. $\Sigma^{?} = \Sigma^{?}_{\mathcal{I}}$ *and* $\Sigma^{!} = \Sigma^{!}_{\mathcal{I}}$*),*

$$\mathcal{I} \textbf{ ioco } \mathcal{M} \quad \triangleq \quad \forall \sigma \in \mathsf{STraces}(\mathcal{M}), Out(\Delta(\mathcal{I}) \texttt{ after } \sigma) \subseteq Out(\Delta(\mathcal{M}) \texttt{ after } \sigma).$$

It can be proved [4] that $\mathcal{I}$ **ioco** $\mathcal{M}$ if and only if $\mathsf{STraces}(\mathcal{I}) \cap \mathsf{MinFTraces}(\mathcal{M}) = \emptyset$, where $\mathsf{MinFTraces}(\mathcal{M}) \triangleq \mathsf{STraces}(\mathcal{M}).\Sigma^{!} \setminus \mathsf{STraces}(\mathcal{M})$ is the set of minimal (with respect to the prefix ordering) non-conformant suspension traces. Notice that the set of all non-conformant traces is then $\mathsf{MinFTraces}(\mathcal{M}).\Sigma^{*}$. This alternative characterisation of **ioco** will be useful in the sequel, in particular for the description of properties of test cases.

Test cases, test suites, properties: In order to complete the formal background, a definition of test cases and test suites (sets of test cases) is provided together with their expected properties relatively to conformance. In practice a test case describes the interaction that should be played when checking conformance of an implementation and the verdicts associated to this interaction. In the present formal setting, the behaviour of a test case is modelled by an IOLTS equipped with colours representing verdicts assigned to executions.

Definition 6 (Test case, test suite)**.** *A* test case *for $\mathcal{M}$ 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 such that $\mathcal{C}_{\mathcal{TC}}$(Pass), $\mathcal{C}_{\mathcal{TC}}$(Fail), $\mathcal{C}_{\mathcal{TC}}$(Inc) and $\mathcal{C}_{\mathcal{TC}}$(None) form a partition of $Q_{\mathcal{TC}}$ and for $\lambda \in \{$Pass, Fail$\}$, $reach_{\mathcal{TC}}(\Sigma, \Lambda) \subseteq \mathcal{C}_{\mathcal{TC}}(\lambda)$ and $reach_{\mathcal{TC}}(\Sigma,$ Inc$) \subseteq \mathcal{C}_{\mathcal{TC}}$(Inc$) \cup \mathcal{C}_{\mathcal{TC}}$(Fail). The alphabet is $\Sigma_{\mathcal{TC}} = \Sigma^{?}_{\mathcal{TC}} \cup \Sigma^{!}_{\mathcal{TC}}$ where $\Sigma^{?}_{\mathcal{TC}} = \Sigma^{!\delta}$ and $\Sigma^{!}_{\mathcal{TC}} = \Sigma^{?}$ (outputs of $\mathcal{TC}$ are inputs of $\mathcal{M}$ 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 $\mathcal{TC} \| \Delta(\mathcal{I})$ where common actions (inputs, outputs and quiescence) are synchronized. The effect is to intersect sets of suspension traces of $\mathcal{I}$ with traces of $\mathcal{TC}$ ($\mathsf{Traces}(\mathcal{TC} \| \Delta(\mathcal{I})) = \mathsf{STraces}(\mathcal{I}) \cap \mathsf{Traces}(\mathcal{TC})$). Consequently, the possible failure of a test case on an implementation is defined as the fact that the interaction of $\mathcal{I}$ and $\mathcal{TC}$ may lead to a state coloured by $Fail$ in $\mathcal{TC}$. Using properties of traces of the parallel composition, this is formalized by $\mathcal{I}$ *fails* $\mathcal{TC} \triangleq \mathsf{STraces}(\mathcal{I}) \cap \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \neq \emptyset$. Notice that $\mathcal{I}$ *fails* $\mathcal{TC}$ only means that $\mathcal{I}$ *may be* rejected by $\mathcal{TC}$, depending on choices made by $\mathcal{I}$ in its interaction with $\mathcal{TC}$. Similar definitions can be given for *passes* and *inconc* relative to the verdicts *Pass* and *Inc*.

Now follows formal definitions of properties that should be satisfied by test cases in order to correctly relate conformance to rejection by a test case:

Definition 7 (Test suites properties)**.** *Let $\mathcal{M}$ be a specification, and $\mathcal{TS}$ a test suite for $\mathcal{M}$.*

- TS *is* sound *if no test case may reject a conformant implementation:*
  $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), (\mathcal{I}$ **ioco** $\mathcal{M} \implies \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{I}$ *fails* $\mathcal{TC}))$.

- $\mathcal{TS}$ *is* exhaustive *if it rejects all non-conformant implementations:* $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), (\neg(\mathcal{I}$ **ioco** $\mathcal{M}) \implies \exists \mathcal{TC} \in \mathcal{TS}, \mathcal{I}$ *fails* $\mathcal{TC})$.

- $\mathcal{TS}$ *is* complete *if it is both sound and exhaustive.*

- $\mathcal{TS}$ *is* strict *if it detects non-conformances as soon as they happen:* $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{TC} \| \mathcal{I}$ **ioco** $\mathcal{M}) \Rightarrow \mathcal{I}$ *fails* $\mathcal{TC}$.

The following characterisations of soundness, exhaustiveness and strictness, derived from [4], are very convenient to prove that generated test suites satisfy those properties. They are obtained by replacing **ioco** by its alternative characterization, *fails* by its definition, replacing universal quantification on $\mathcal{TC}$ by a union, and suppressing the universal quantification on $\mathcal{I}$, using an argument on sets to replace implication by inclusion.

Proposition 1 ([4])**.** *Let $\mathcal{TS}$ be a test suite for $\mathcal{M}$,*

- $\mathcal{TS}$ *is sound if* $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}) \subseteq \mathsf{MinFTraces}(\mathcal{M}).\Sigma^*$,

- $\mathcal{TS}$ *is exhaustive if* $\mathsf{MinFTraces}(\mathcal{M}) \subseteq \bigcup_{\mathcal{TC} \in \mathcal{TS}} \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC})$,

- $\mathcal{TS}$ *is strict if* $\forall \mathcal{TC} \in \mathcal{TS}, \mathsf{Traces}(\mathcal{TC}) \cap \mathsf{MinFTraces}(\mathcal{M}) \subseteq \mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC})$.

(According to earlier notations, the set $\mathsf{Traces}_{\mathcal{C}_{\mathcal{TC}}(Fail)}(\mathcal{TC}))$ is formed by the traces in $\mathcal{TC}$ leading to a state with colour *Fail*.)

Informally, soundness is characterized by the fact that traces of test cases leading to *Fail* are non-conformant traces. Exhaustiveness means that all non-conformant traces are recognized in *Fail* states of some test case. Furthermore, strictness means that traces of test cases which are minimal non-conformant ones lead to a *Fail* state.

3 Recursive Tile Systems and their properties

This section provides a definition for the *Recursive Tile Systems* (RTSs), a model finitely representing infinite state IOLTSs inspired from the regular graphs of Courcelle [10]. These systems form classical middle-ground between finite state systems and Turing-complete ones. They are expressive enough to model recursive systems, yet many properties remain decidable. RTSs in particular are graphical finite representations, as such they seem simple and intuitively close to finite IOLTSs. The present section introduces some algorithms and properties of RTSs: ε-closure (suppression of internal actions), product and determinization. These properties are useful for test generation which will be discussed in the next section.

3.1 Recursive tile systems

Roughly speaking an RTS is a finite collection of finite transition systems (called *tiles*) together with identifications enabling to connect these tiles. Each RTS generates a single *infinite* IOLTS composed of finite patterns which correspond to the tiles.

Definition 8 (Recursive tile system)**.** *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 $t_\mathcal{A} = ((\Sigma, \Lambda), Q_\mathcal{A}, \rightarrow_\mathcal{A}, \mathcal{C}_\mathcal{A}, S_\mathcal{A}, F_\mathcal{A})$ 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,
- $S_A \subseteq Q_A$ *is the* support
- $F_{\mathcal{A}} \subseteq \mathcal{T} \times 2^{\mathbb{N}\times\mathbb{N}}$, *the* frontier, *relates to some* tile, $t_{\mathcal{B}}$, *a partial function (often denoted* $f_{\mathcal{B}}$*) over* $\mathbb{N}$, *associating to each vertex of the support* $S_{\mathcal{B}}$, *vertices of* $Q_{\mathcal{A}}$.

- $t_0 \in \mathcal{T}$ *is an initial tile (the axiom), with* $S_0 = \emptyset$.

Each single tile $t_{\mathcal{A}}$ defines an IOLTS $[t_{\mathcal{A}}] = (Q_{\mathcal{A}}, \Sigma, \Lambda, \rightarrow_{\mathcal{A}}, \mathcal{C}_{\mathcal{A}}, \texttt{init})$ in a straightforward way when ignoring the support and frontier.

The tiling operation (that will be defined later on) inductively constructs an IOLTS from an RTS. The support of a tile indicates vertices which may be attached to other tiles, in the tiling operation. The frontier $F_{\mathcal{A}}$ of a tile $t_{\mathcal{A}}$ defines which tiles $t_{\mathcal{B}}$ are attached to $t_{\mathcal{A}}$ along the tiling, it also specifies how vertices of the support of $t_{\mathcal{B}}$ are merged with vertices of $t_{\mathcal{A}}$ by this operation.

The V*-frontier* of any tile $t_{\mathcal{A}}$, is the set formed by the vertices which belong to the image of any function in the frontier, formally, V-frontier $= \bigcup_{\{f_{\mathcal{B}} \mid (t_{\mathcal{B}}, f_{\mathcal{B}}) \in F_{\mathcal{A}}\}} Im(f_{\mathcal{B}})$.

Example 1. *The program depicted on Fig. 1, may be abstracted in the following RTS:* $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_{\texttt{main}})$ *with* $\Sigma^{\tau} = \{\texttt{try}, \texttt{throw}, \texttt{catch}, \texttt{intern}\}$, $\Sigma^{?} = \{\texttt{int}, \texttt{true}, \texttt{false}\}$, $\Sigma^{!} = \{\texttt{m1}, \texttt{m2}, \texttt{m3}, \texttt{m4}\}$, $\Lambda = \{\texttt{init}, \texttt{succ}\}$, *a set of tiles* $\mathcal{T} = \{t_{\texttt{main}}, t_{\texttt{comp}}\}$ *(a graphical presentation of these tiles is depicted on Fig. 2), and* $t_{\texttt{main}}$ *the initial tile. The output actions correspond to messages:* `m1` *is* `Done`, `m2` *is* `An error has occurred`, `m3` *is* `Some text` *and* `m4` *is* `You stopped`. *The symbol* `int` *stands for the integer input, and observe that the actual value of this input is not reflected by the structure of the RTS, inputs* `true`, `false` *reflect the boolean input in block 4. The symbol* `intern` *reflects the unlabelled internal action in block 5 (the computation).*

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

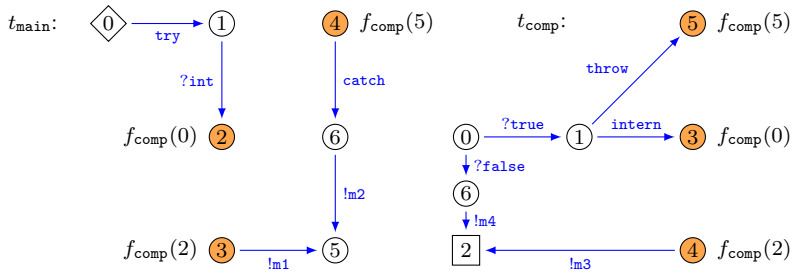

Figure 2: Tiles composing the system $\mathcal{R}$

For the frontier, e.g., in the tile $t_{\texttt{main}}$, $f_{\texttt{comp}}(0)$ (2) *means that* $(t_{\texttt{comp}}, \{0 \rightarrow 2\})$ *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 $t_{\mathcal{E}} = ((\Sigma, \Lambda), Q_{\mathcal{E}}, \rightarrow_{\mathcal{E}}, \mathcal{C}_{\mathcal{E}}, S_{\mathcal{E}}, F_{\mathcal{E}})$ 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 $t'_{\mathcal{E}} = ((\Sigma, \Lambda), Q'_{\mathcal{E}}, \rightarrow'_{\mathcal{E}}, \mathcal{C}'_{\mathcal{E}}, S'_{\mathcal{E}}, F'_{\mathcal{E}})$ iteratively defined according to the elements of the frontier $F_{\mathcal{E}}$, as follows:

1. Initially, $Q'_\varepsilon = Q_\varepsilon$, $\rightarrow'_\varepsilon = \rightarrow_\varepsilon$, $\mathcal{C}'_\varepsilon = \mathcal{C}_\varepsilon$, $S'_\varepsilon = S_\varepsilon$, $F'_\varepsilon = \emptyset$;
2. For each pair $(t_\mathcal{B}, f_\mathcal{B}) \in F_\varepsilon$, with $t_\mathcal{B} = ((\Sigma, \Lambda), Q_\mathcal{B}, \rightarrow_\mathcal{B}, \mathcal{C}_\mathcal{B}, S_\mathcal{B}, F_\mathcal{B}) \in \mathcal{T}_\mathcal{B}$,
   let $\varphi_\mathcal{B} : Q_\mathcal{B} \rightarrow \mathbb{N}$ be the injection mapping vertices of $Q_\mathcal{B}$ to new vertices of Q'_ε with $\varphi_\mathcal{B}(n) := f_\mathcal{B}(n)$ whenever $n \in dom(f_\mathcal{B}), n + \max(Q'_\varepsilon) + 1$ otherwise, where $\max(Q'_\varepsilon)$ is the vertex with greatest value in Q'_ε. The tile t'_ε is then defined by:
   - $Q'_\varepsilon = Q'_\varepsilon \cup Im(\varphi_\mathcal{B})$,
   - $S'_\varepsilon = S'_\varepsilon$,
   - $\rightarrow'_\varepsilon = \rightarrow'_\varepsilon \cup \{(\varphi_\mathcal{B}(n), a, \varphi_\mathcal{B}(n')) \mid (n, a, n') \in \rightarrow_\mathcal{B}\}$,
   - $\mathcal{C}'_\varepsilon = \mathcal{C}'_\varepsilon \cup \{(\varphi_\mathcal{B}(n), \lambda) \mid (n, \lambda) \in \mathcal{C}_\mathcal{B}\}$,
   - $F'_\varepsilon = F'_\varepsilon \cup \{(t_\mathcal{C}, \{(\varphi_\mathcal{B}(j), f_\mathcal{C}(j)) \mid j \in dom(f_\mathcal{C})\}) \mid (t_\mathcal{C}, f_\mathcal{C}) \in F_\mathcal{B}\}$. 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 (the same tiles up to renaming of the vertices). More precisely, one could define an order on the way to append tiles from the frontier and thus produce a single possible tile after the tiling. This process would be long and intricate. The benefit would be limited since every possible order produces an isomorphic tile (up to a renaming of vertices). Hence, the choice, here, is not to fix this order, enabling the production of several isomorphic semantics for a given RTS.*

Example 2. *The principle of tiling is illustrated now, using the RTS defined in Example 1. Consider that $t_{\texttt{main}}$ is the initial tile (it has empty support). Its tiling $\mathcal{T}(t_{\texttt{main}})$, is performed as follows: there is a single element in its frontier; a copy of $t_{\texttt{comp}}$ (with new vertices) is added, identifying vertices $2, 3$ and 4 of $t_{\texttt{main}}$ to vertices $0, 2$ and 5 (the support) of $t_{\texttt{comp}}$.*

The resulting tile is depicted in Fig. 3 (top). This new tile may be in turn extended by adding a copy of $t_{\texttt{comp}}$, identifying $4, 10$ and 11 respectively to $0, 2$ and 5. Again, the resulting tile is illustrated in Fig. 3 (bottom) (observe that the definition of $\varphi_{\texttt{comp}}$ 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 IOLTSs of tiles resulting from the iterated tilings from the axiom. Formally,

Definition 9 (Semantic of an RTS)**.** *Let $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ be an RTS. $\mathcal{R}$ defines an IOLTS $[\![\mathcal{R}]\!] = (Q_\mathcal{R}, \Sigma, \Lambda, \rightarrow_\mathcal{R}, \mathcal{C}_\mathcal{R}, \texttt{init})$ given by*

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

The infinite union of Definition 9 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 IOLTSs, *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 number of tiling operations needed to create q, formally, the least $k \in \mathbb{N}$ such that q is a state of $[\mathcal{T}^k(t_0)]$. Also, $t(q)$ 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 computations, some technical restrictions are imposed on the RTS, $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, that can be ensured by a normalization[3] 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, one may assume that each vertex possesses a colour reflecting this value (see Corollary 1 below).*

[3] Such a normalization transforms the tiles of an RTS without changing its semantic.

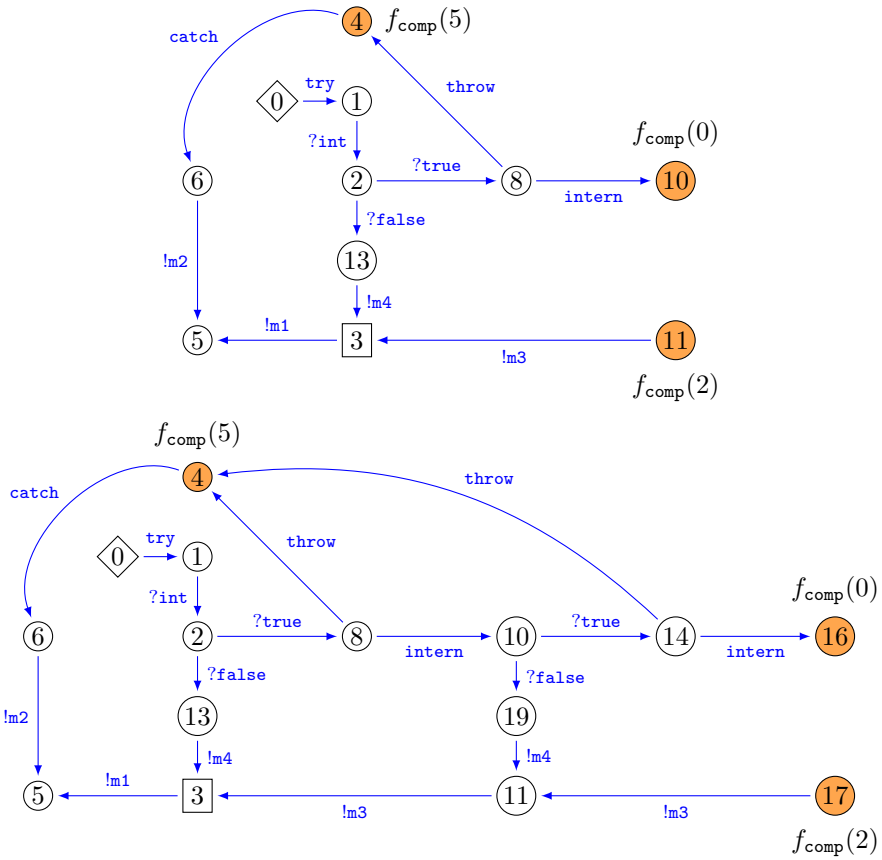

Figure 3: $\mathcal{T}(t_{\texttt{main}})$ and $\mathcal{T}^2(t_{\texttt{main}})$ tiles

The first restriction will simplify the computation of the Σ^τ-closure (this operation will be defined in Section 3.3), and the second will simplify the computation of the suspended specification.

Remark 2. *The IOLTSs obtained from RTSs correspond to the* equational*, or* regular *graphs of respectively Courcelle and Caucal [10, 11]. These IOLTSs are derived from an axiom using deterministic* HR*-grammars. Each such grammar may be transformed into a tiling system, and conversely. Deterministic* HR*-grammars are defined by sets of graph-rewriting rules. Each left-hand side is formed by a non-terminal hyperedge (corresponding to the notion of support in tiles). Each right-hand side is formed by a finite hypergraph. In this hypergraph, the set of non-terminal hyperedges corresponds to the frontier, terminal hyperedges are ordinary transitions. This definition of RTSs aims at a greater simplicity by focusing the definition on a finite set of graphs rather than a finite set of rules, and removing hyperedges which are only a syntactical element used to connect tiles.*

3.2 Reachability of RTSs

This subsection presents fundamental results on RTSs with respect to the formal generation of test suites. In particular to detect quiescent states, and to prune the canonical tester.

Reachability.

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

Proposition 2 ([11])**.** *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 $\mathcal{R}'$ 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: $\mathcal{C}_{[\![\mathcal{R}']\!]}(r_\lambda) = reach_{[\![\mathcal{R}]\!]}(\mathcal{C}_{[\![\mathcal{R}]\!]}(\lambda), \Sigma')$. The same result holds for states co-reachable from λ.*

Proposition 2 is established, for regular graphs, by Caucal [11], the resulting system may be of exponential size in the size of the largest support. However, in the following we will first use a normal form (of quadratic size), such that the size of the support of each tile will have size 2. Hence, with respect to the computations we are performing, reachability computation takes polynomial time.

Now, in Caucal [11], Proposition 3.13 enables to perform several computations related to the purpose of this paper. The following proposition is a reformulation for RTSs.

Proposition 3 ([11])**.** *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 in S is coloured by $\#_S$.*

This computation may be performed in linear time: in a given tile t the neighbourhood of each vertex, not in its support, only depends on t and possibly the tiles in its frontier.

Proposition 3 enables to identify directly on the set of tiles some state properties, like deadlocks, inputlocks. The following corollary is also a direct consequence of this proposition (performing successively, for each action a, a colouring for the degree related to a).

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

3.3 Σ^τ-closure of RTSs

Abstracting away internal transitions (labelled by actions in Σ^τ) is important to compute the next observable actions after a trace, thus for test generation. While the following proposition shows it is possible to do it for RTSs, the rest of the subsection will be devoted to establish a more precise result (Proposition 5) and will provide an algorithm to perform the Σ^τ-closure of RTSs.

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 for pushdown systems and regular graph; it follows mainly from the work of Caucal [11] and can be adapted to RTSs: from a given RTS (labelled by $\Sigma^\tau \cup \Sigma^o$), a context-free grammar generating the same set of traces (in Σ^{o*}) may be constructed, then from such a grammar an RTS of finite degree may be constructed.

Now follows a direct construction, in order to provide an accurate evaluation of the complexity of the process, and to assess which properties of the original RTS are preserved. In particular our construction will take colours into account since it is essential for the generation of tests cases. First, a careful examination of which equivalence between IOLTSs may be considered, is performed. Then, a specific computation of the Σ^τ-closure of an IOLTS (preserving this equivalence) is proposed. Since this new computation is not effective for infinite state systems, a direct transformation on RTSs (performing this computation for the generated IOLTSs) is given.

3.3.1 Equivalence for IOLTSs.

The computation of a closure for transition systems is usually focused on traces preservation, and performed either by forward or backward computation. Since, in this paper, the states of IOLTSs have colours, such a straightforward computation may result in the loss of important information, indeed, from Proposition 3, deadlocks, outputlocks, hence quiescence are specified by colours. Thus, a notion of coloured traces is introduced in order to obtain a finer equivalence, preserving this information.

Coloured traces, and coloured equivalence. A *coloured trace* is a finite sequence in $(\Lambda.(\Sigma^o)^+)^+.\Lambda$. A coloured trace $\lambda_1 w_1 \lambda_2 w_2 ... \lambda_n$ is *recognized* by an IOLTS $\mathcal{M} = (Q, \Sigma, \Lambda, \rightarrow, \mathcal{C}, \texttt{init})$ if there exists n states $q_1, q_2, ..., q_n$ such that for all $1 \leq k \leq n-1$, $q_k \stackrel{w_k}{\Longrightarrow} q_{k+1}$, and for all $1 \leq k \leq n$, $q_k \in \mathcal{C}(\lambda_k)$.

Observe that paths in this definition do not necessarily start from a state coloured by `init`. Moreover, given a path in some IOLTS, several distinct coloured traces might be defined from this same path since a state can have several colours. Finally, the empty word may not label a coloured trace, this is an arbitrary choice guided by a technical reason: preserving ε-labelled coloured traces would be much more difficult and impose a much more complex definition of colours in states.

Definition 10 (Coloured equivalence)**.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be two IOLTSs. $\mathcal{M}$ and $\mathcal{M}'$ are* coloured equivalent *whenever they recognise the same coloured traces.*

The coloured equivalence is more precise than trace equivalence, since two coloured equivalent systems have the same traces (up to the empty word) whereas the converse if not true in general. Conversely coloured equivalence is less precise than bisimulation.

3.3.2 Mixed closure.

The purpose is eventually to compute the closure, with respect to internal events in Σ^τ, of IOLTSs defined by RTSs. A naive approach to accomplish such a computation would be to perform it in each tile. Unfortunately both forward and backward closures face difficulties to deal with states generated at the V-frontier of some tile. Hence the introduction of a general process: *mixed closure* which will be suited to RTSs and furthermore will preserve coloured traces. This subsection presents the principles of mixed closure for IOLTSs and the next one will present its adaptation to systems generated by RTSs.

Roughly speaking this approach simply consists in adding a new state for each pair of states connected by a Σ^τ-labelled transition (Σ^τ-transition for short), connecting this new state to each predecessor of the source of the transition and to each successor of the target. Whenever there exist strongly connected components labelled by actions in Σ^τ, this process will proceed forever. Hence this technique will first eliminate these cycles.

In order to present the mixed closure, consider an IOLTS $\mathcal{M} = (Q, \Sigma, \Lambda, \rightarrow_{\mathcal{M}}, \mathcal{C}, \texttt{init})$ having Σ^τ labelled transitions. Let $Clo(\mathcal{M}) = (Q_{Clo}, \Sigma, \Lambda, \rightarrow_{Clo}, \mathcal{C}_{Clo}, \texttt{init})$ be the resulting mixed closure of $\mathcal{M}$. This system is obtained after several iterations constructing intermediate IOLTSs denoted by $\mathcal{M}_i$ with sets of states, transitions and

colouring denoted respectively by $Q_i, \rightarrow_i, \mathcal{C}_i$ (observe that the sets of labels and of colours are not modified). We now start by considering strongly connected components of Σ^τ-transitions, then we will perform the closure of remaining unobservable transitions.

Strongly connected component of Σ^τ-transitions. Let $\{C_1, \ldots, C_N\}$ be the set of maximal strongly connected components (SCCs) of Σ^τ-transitions in $\mathcal{M}$. Let $\mathcal{M}_0 \triangleq \mathcal{M}$, then, for all $i = 1 \ldots N-1$, the states, transitions, and colouring of $\mathcal{M}_{i+1}$ are as follows:

- $Q_{i+1} \triangleq Q_i \cup \{\hat{q}_i\}$ with $\hat{q}_i$ a new state;
- For all states q of Q_{i+1}, $\mathcal{C}_{i+1}(q) = \bigcup_{p \in C_i} \mathcal{C}_i(p)$ if $q = \hat{q}_i$, $\mathcal{C}_{i+1}(q) = \mathcal{C}_i(q)$ otherwise;
- $$\begin{aligned} \rightarrow_{i+1} \triangleq \Big(\rightarrow_i \setminus \{(q,\mu,q') \mid q, q' \in C_i\} \quad &\cup \quad \{(q,\mu,\hat{q}_i) \mid \exists q_\tau \in C_i, \mu \in \Sigma, (q,\mu,q_\tau) \in \rightarrow_i\} \\ &\cup \quad \{(\hat{q}_i,\mu,q) \mid \exists q_\tau \in C_i, \mu \in \Sigma, (q_\tau,\mu,q) \in \rightarrow_i\} \Big). \end{aligned}$$

For all $i = 1 \ldots N-1$, $\mathcal{M}_{i+1}$ has the same coloured traces as $\mathcal{M}_i$ (hence same as $\mathcal{M}$) since paths having no transitions in Σ^τ are preserved and those having such transitions may only have at most one colour of a sequence in $(\Sigma^\tau)^+$ and each such colour is kept.

Closure of remaining Σ^τ-transitions. Assume that $\mathcal{M}$ has been transformed into $\mathcal{M}_N$ (where N is the total number of SCCs in $\mathcal{M}$), and thus has no strongly connected component of Σ^τ. Now, internal transitions are iteratively suppressed by defining new IOLTS $\mathcal{M}_{N+1}, \ldots, \mathcal{M}_{N+k}$ as follows, starting from $M_i = M_N$.

If there exists a transition $q_1 \xrightarrow{\tau} q_2 \in \rightarrow_i$ for some $\tau \in \Sigma^\tau$, then the states, transitions, and colouring of $\mathcal{M}_{i+1}$ are as follows:

- $Q_{i+1} \triangleq Q_i \cup \{\hat{q}_{12}\}$ with $\hat{q}_{12}$ a new state;
- For all states q of Q_{i+1}, $\mathcal{C}_{i+1}(q) = \mathcal{C}_i(q_1) \cup \mathcal{C}_i(q_2)$ if $q = \hat{q}_{12}$, $\mathcal{C}_{i+1}(q) = \mathcal{C}_i(q)$ otherwise;
- $$\begin{aligned} \rightarrow_{i+1} \triangleq \Big(\rightarrow_i \setminus \{(q_1,\tau,q_2)\} \quad &\cup \quad \{(q,\mu,\hat{q}_{12}) \mid q \in Q_i, \mu \in \Sigma, (q,\mu,q_1) \in \rightarrow_i\} \\ &\cup \quad \{(\hat{q}_{12},\mu,q) \mid q \in Q_i, \mu \in \Sigma, (q_2,\mu,q) \in \rightarrow_i\} \Big). \end{aligned}$$

For all $i \geq N$, $\mathcal{M}_{i+1}$ has the same coloured traces as $\mathcal{M}_i$ (hence same as $\mathcal{M}$) since the only transformation is to replace a single internal transitions by a single state having both colours of the states connected by this transition (and all in-transitions of the sources, and all out-transitions of the target), and coloured traces have at least one visible action between two consecutive colours. Eventually the closure of $\mathcal{M}$, $Clo(\mathcal{M})$, has the same coloured traces as $\mathcal{M}$.

Whenever the system $\mathcal{M}$ has finitely many states, the resulting system $Clo(\mathcal{M})$ is obtained after finitely many steps (reducing the length of a sequence of internal transitions at each step). In general this may not be applied for infinite state IOLTSs.

3.3.3 Effective mixed closure for RTSs.

Even though the construction of the mixed closure is not effective for infinite state systems in general, it is possible to adapt this construction for RTSs by transforming the tiles of an RTS and construct another coloured equivalent RTS. Observe, also, that a non-careful transformation could produce states of infinite degree which is often not desirable. The transformation presented here will avoid producing such states.

The most naive approach to compute mixed closure for a system generated by a RTS would be to proceed for each tile independently. This idea fails whenever some internal transitions are connected to the support or to a vertex of the V-frontier of a tile. It fails even more blatantly when a sequence of such transitions connects a vertex of the support to one of the V-frontier. In order to solve these problems, a normal form (introduced in Hassen's PhD thesis [15, 12]) is presented and followed by a technique that removes paths of internal transitions traversing tiles from the support to the V-frontier (or conversely). The final step consists in iterating a finite closure in each tile.

Path tiles. *Path tiles* is a normal form of RTS which focuses on paths. This normal form transforms the generated IOLTS by duplicating some paths while preserving coloured traces. It allows a simple application of the mixed closure assuming that there are no paths of Σ^τ actions traversing any tile from support to V-frontier.

A *path tile* is a tile whose support is composed of at most one vertex having null in-degree and at most one with null out-degree.

Given any RTS it is possible to construct a coloured-equivalent RTS having only path tiles. This construction is straightforward and consists simply, for any original tile $t_{\mathcal{A}}$, and any pair of vertices v and v' (which may not be distinct) in the support of t, in constructing a new tile $t_{\mathcal{A}vv'}$ having v and v' as support (when they are identical two distinct vertices will be present in $t_{\mathcal{A}vv'}$), and respectively v and v' with null in and out-degree. The tile $t_{\mathcal{A}vv'}$ contains only the out-transitions from v, and in-transitions to v', and vertices of t reachable from v and co-reachable from v'. The frontier is defined accordingly splitting each tile in the frontier into each of its components (according to this decomposition). Two other tiles, $t_{\mathcal{A}vo}$ and $t_{\mathcal{A}vi}$, are defined for v, having respectively null out- and in-degree, and containing only, respectively, states reachable and co-reachable from v (the frontier is built similarly).

This transformation duplicates several states but enables complete preservation of coloured traces. Furthermore it produces a quadratic number of tiles (in the cardinality of the supports).

Removing Σ^τ-paths between support and V-frontier. There are two symmetrical operations. Only the one removing internal transitions from the support to the V-frontier is presented here.

From the previous construction, without loss of generality, one may assume that the RTS $\mathcal{M}$ is only formed of path tiles (for the sake of simplicity the same convention as in the previous paragraph is assumed for the name of those tiles).

Let $t_{\mathcal{A}vv'}$ be a tile of $\mathcal{M}$ having Σ^τ-paths from v to elements of the V-frontier. The following step is iterated to construct a new tile $t'_{\mathcal{A}vv'}$ which will not have any traversing Σ^τ-paths, or traversing paths reaching tiles that have already been traversed:
For each v'' in the V-frontier of $t_{\mathcal{A}vv'}$, with $(t_{\mathcal{B}ww'}, v_{\mathcal{B}}, v'') \in F_{\mathcal{A}}$, target of a Σ^τ-path from v, tile $t_{\mathcal{A}vv'}$ with $t_{\mathcal{B}ww'}$.

Then, for each vertex, $v_{\mathcal{B}}$, of the V-frontier, connected to some tile $t_{\mathcal{B}ww'}$ which is target of a Σ^τ-path, identify the vertex, $v'_{\mathcal{B}}$, corresponding to the previous occurrence of $t_{\mathcal{B}ww'}$ (which is always possible from the halting condition of the previous iteration).

For each transition $v \xrightarrow{\mu} v_{\mathcal{B}}$, with $\ell(v) \geq \ell(v_{\mathcal{B}})$ (appearing in or after the occurrence of $t_{\mathcal{B}ww'}$), add a transition $v \xrightarrow{\mu} v'_{\mathcal{B}}$. When all these transitions have been added, remove v from the frontier of the constructed tile.

Iterating the previous process for each tile reached by a Σ^τ-path, the resulting tile, denoted by $t'_{\mathcal{A}vv'}$ has no traversing Σ^τ-path. Iterating this process on each tile produces a coloured equivalent RTS with no Σ^τ-path.

The removal of paths from the V-frontier to the support is performed similarly, except that the transitions are considered the opposite way.

In order to perform these operations, an exponential number of tiles may be appended to each original tile. Every path in a tree need to be considered to remove traversing Σ^τ-paths. Note, however, that in order to reach such an important size a considerable amount of traversing Σ^τ-paths needs to exists in the system.

Removing internal paths. Now in order to perform the mixed closure, first SCCs of internal transitions need to be identified. Since there are no traversing Σ^τ-paths, each SCC either fully belongs to a tile or belongs to adjacent tiles. Hence the closure of internal transitions between elements of the support may be performed: given a tile $t_{\mathcal{A}}$ having paths of internal transitions between elements of its support, extra vertices are added to the support of $t_{\mathcal{A}}$ and tiles having $t_{\mathcal{A}}$ in their supports are modified accordingly. Once it has been performed, the converse is done for paths of internal transitions between vertices. Afterwards, for every tile, a mixed closure is applied inside the tile.

First, removal of SCCs may be performed in polynomial time, in the number of vertices. Then, the mixed closure is exponential: for each sequence of silent actions of length n (n is smaller than the number of vertices), $(n+1)n/2$ new vertices may be constructed. In the worst case, there is an exponential number of such sequences hence an exponential bound.

Proposition 5. *From an RTS $\mathcal{R}$ with internal actions Σ^τ, one can effectively compute an RTS $Clo(\mathcal{R})$ of exponential size in the number of vertices, such that its semantics $[\![Clo(\mathcal{R})]\!]$ has no internal action, is of finite* out*-degree, and has the same coloured traces as the IOLTS $[\![\mathcal{R}]\!]$.*

From earlier observations the complete process of performing the closure is performed in exponential time.

Weighted RTSs.

In formal testing, off-line test computations are performed from deterministic models, in order to determine all possible outputs after any trace. However RTSs are not determinizable in general, in fact, Section 5 presents an on-line approach to perform tests on RTSs which one do not wish to determinize. Still there are some RTSs which may be determinized, in particular, weighted RTSs form a decidable class of RTSs which have an effective determinization process.

Definition 11 (Weighted RTS)**.** *An RTS $\mathcal{R}$ with no internal action and with IOLTS semantics $[\![\mathcal{R}]\!] = (Q, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}, \texttt{init})$ is* weighted *if $\mathcal{C}(\texttt{init})$ is a singleton $\{q_0\}$, and for any $w \in \Sigma^*$ and any pair of states $q, q' \in Q$, $q_0 \xrightarrow{w} q$ and $q_0 \xrightarrow{w} q'$ implies $\ell(q) = \ell(q')$: two states reached by the same sequence have the same level*[4]*.*

Determining whether a given RTS is weighted is decidable (Lemma 4.1 by Caucal and Hassen [12]), in polynomial time. The algorithm initially provided in Hassen's thesis [15] for HR-grammars can be explained for RTSs as follows. The computation is performed by three successive fixed-points which do not modify the set of tiles (hence the polynomial bound). The first one consists in computing the set of outgoing labels for vertices of the support of tiles (linear computation). The second fixed-point is the computation of sets of pairs of vertices (in a corresponding product of tiles) which are synchronized in the (formal) product RTS. Synchronized means that a given sequence of actions starting from each vertex of the pair reaches vertices of the same depth. This computation is polynomial since there are a quadratic number of pairs, each one is connected to a set of at most a quadratic number of it. The last fixed-point is defining the set of synchronized vertices (in the original RTS), building from the previous step, thus producing a smaller set. Whenever the last set witnesses a vertex which belongs to the support of a tile synchronized with a vertex that does not, the RTS is not weighted. Hence a globally polynomial decision process.

Determinization of RTSs.

An RTS $\mathcal{R}$ is said *deterministic* if its underlying IOLTS $[\![\mathcal{R}]\!]$ is deterministic. This property is decidable from the set of tiles defining it (for example using Proposition 3). However, since PDAs cannot be determinized in general, there is no hope to determinize an arbitrary RTS. Still, there are some classes of determinizable PDAs, like visibly PDAs [16], or, more recently, *weighted grammars* [17]. These grammars define a class of PDAs that can be determinized and which both subsume the visibly PDAs and the height deterministic PDAs [18].

Proposition 6 ([12])**.** *Any weighted RTS $\mathcal{R}$ (with no internal transition) 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.*

This operation implies first to transform the RTS, so that it only has path-tiles, then these tiles are merged with respect to the vertices with positive out-degree. Both these computations may be performed in polynomial time. Eventually the computation is performed inside tiles resulting in an exponential complexity similar to the case of finite state systems.

Example 3. *Assuming internal actions are not visible, the RTS defined in Example 1 is slightly modified: assume that vertex 5 is not in the V-frontier anymore, and suppose that there are 3 transitions labelled* `int` *between 0 and respectively 1, 3 and 5. The resulting system is weighted. In such a situation, determinization would simply perform a finite IOLTS determinization in the tile $t_{\texttt{comp}}$. In the general case some tiles need first to be merged.*

Synchronous product.

As seen in Section 2.1, the synchronous product of IOLTSs is the operation used to intersect languages. It is also useful for test selection using a test purpose. In general, the model recognizing the intersection of languages of two RTSs is not recursive. Indeed, the intersection of two context-free languages can be obtained by a product of two RTSs, if such a product was recursive the intersection of two context-free languages would be a context-free language

[4] Are generated after the same number of tilings.

(*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). However, 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 IOLTSs). This RTS is defined as follows; let $\mathcal{A} = (Q, \Sigma, \Lambda, \rightarrow_{\mathcal{A}}, \mathcal{C}, \texttt{init})$ be a finite IOLTS, and $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ be a RTS. The set of tiles $\mathcal{T}_{\mathcal{R}\times\mathcal{A}}$ is the sets of products of the tiles of $\mathcal{T}$ in synchronous product with $\mathcal{A}$. Formally, for a given tile $t_{\mathcal{B}} \in \mathcal{T}$, with $t_{\mathcal{B}} = ((\Sigma, \Lambda), Q_{\mathcal{B}}, \rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, S_{\mathcal{B}}, F_{\mathcal{B}})$, the product tile, denoted by $t_{\mathcal{B}} \times \mathcal{A}$, is the following: $t_{\mathcal{B}} \times \mathcal{A} = ((\Sigma, \Lambda \times \Lambda), Q_{\mathcal{B}} \times Q, \rightarrow_{\mathcal{B}\times\mathcal{A}}, \mathcal{C}_{\mathcal{B}\times\mathcal{A}}, S_{\mathcal{B}\times\mathcal{A}}, F_{\mathcal{B}\times\mathcal{A}})$, with the transitions and colours defined like for products of IOLTSs in Section 2.1, the support is simply: $S_{\mathcal{B}} \times Q$, and, for each $(t_{\mathcal{C}}, f_{\mathcal{C}}) \in F_{\mathcal{B}}$ ($f_{\mathcal{C}} : S_{\mathcal{C}} \rightarrow Q_{\mathcal{B}}$), there is a $(t_{\mathcal{C}\times\mathcal{A}}, f_{\mathcal{C}\times\mathcal{A}})$ with $t_{\mathcal{C}\times\mathcal{A}}$ another tile of the product, and $f_{\mathcal{C}\times\mathcal{A}}$ a function between $S_{\mathcal{C}} \times Q$ and $Q_{\mathcal{B}} \times Q$ that associates to any pair $(q_{\mathcal{C}}, q)$ the pair: $(f_{\mathcal{C}}(q_{\mathcal{C}}), q)$. Any coloured trace of the product may be projected (with respect to colours) on either one of the systems and is a coloured trace of this system.

3.4 Effective run execution in RTSs

In order to perform passive testing (testing which does not involve providing input to the implementation), monitoring, diagnosis or on-line testing, one needs to follow an actual execution on a model of the specification. Whenever such a specification is given by an RTS it is not necessary to actually construct recursively the tilings in order to follow symbolically an execution, or to check whether some observed run is a correct execution of the system.

Runs in Deterministic RTSs.

Given a deterministic RTS $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$, and a word $w = \mu_0 \cdots \mu_{n-1} \in \Sigma^*$, let $v_0 \in t_0$ be such that $v_0 \in \texttt{init}$. Let T be a set of arbitrary symbols denoting tiles and let π be the bijection mapping tiles of $\mathcal{T}$ into symbols of T. The *symbolic path* labelled by w is the following sequence: $(v_0, \pi(t_0)) \xrightarrow{\mu_0} (v_1, u_1) \cdots (v_i, u_i) \xrightarrow{\mu_i} (v_{i+1}, u_{i+1}) \cdots \xrightarrow{\mu_{n-1}} (v_n, u_n)$, each word u_i is a sequence of symbols in T representing tiles traversed in the past, and each v_i is a vertex in some tile (the inverse image by π of the last symbol in u_i). It is assumed here that, either in any tile at most one tile of each kind may be tiled, or the set of symbols, T, enables unambiguous identification of the precise traversed tiles (for example having indices to distinguish several occurrences of a tile in the frontier of another). For each i, transition $(v_i, u_i) \xrightarrow{\mu_i} (v_{i+1}, u_{i+1})$ corresponds to one of the three cases (assume $v_i \in t_i$ for tile t_i which is the last in u_i: $t_i = \pi^{-1}(u_i(|u_i| - 1))$) described hereafter. Observe that these three operations correspond respectively to the internal, pop and push operations of pushdown automata:

- (internal) the transition labelled μ_i belongs to tile t_i, then v_{i+1} is simply the target of this transition and $u_{i+1} = u_i$;
- (pop) the transition labelled μ_i reaches the support of t_i, then use the frontier of the tile $\pi^{-1}(u_i(|u_i| - 2))$ to identify the state v_{i+1} corresponding to it. Then u_{i+1} is formed by the first $|u_i| - 1$ symbols of u_i;
- (push) the vertex v_i belongs to the frontier (and there is no transition labelled μ_i in t_i), then assuming that t_{i+1} is the tile containing a transition labelled μ_i starting at the inverse image of v_i in the frontier. First, let state v_{i+1} be the image of such transition in tile t_{i+1}, and second, $u_{i+1} = u_i \pi(t_{i+1})$.

Whenever the RTS is deterministic, there is at most one symbolic path corresponding to a word. The computation of this symbolic path does not require to compute the whole system the actual path traverses.

Runs in Non-deterministic RTSs and Weighted RTSs.

In the case of a non-deterministic RTS, a word w in Σ^* may label several symbolic paths from the states labelled `init`. In fact there may be exponentially many such paths (with respect to the length of w). Furthermore there is no guaranty on the words of T^* representing sequences of tiles: these words may evolve completely independently, reaching any length between 0 and the length of w. On the other hand, Weighted RTSs may also produce exponentially many symbolic paths for a given word w in Σ^* (this is unavoidable and may also occur for finite state systems). But each symbolic word in T^*, reached by w, will have same length, enabling efficient representation and computation of continuations.

Note on Implementation.

In order to efficiently implement runs in systems modelled by RTSs, the system only needs to have access to a single copy of each tile. For each symbolic path, a pair formed by the current vertex and a word of T^* must be kept. This data structure may become large but it is much smaller than the actual tile obtained after k tilings when k is large.

4 Off-line test generation for weighted RTSs

This section and the following consider the generation of test cases from RTSs. The present section focuses on weighted RTSs, which are determinizable, and proposes an off-line test generation algorithm that operates a selection guided by a test purpose (specified by a finite IOLTS). For a finite IOLTS $\mathcal{M}$, off-line test generation guided by an IOLTS test purpose $\mathcal{TP}$ consists in a series of operations as follows (see *e.g.* the work of Jard and Jéron [2]): first the suspension IOLTS $\Delta(\mathcal{M})$ is computed, and determinized into an IOLTS $deter(\mathcal{M})$. Next, this IOLTS is completed by directing unspecified outputs into *Fail* states, and mirroring actions, giving rise to the so called *canonical tester* $Can(\mathcal{M})$. Then, the product IOLTS $Can(\mathcal{M}) \times \mathcal{TP}$ is computed, allowing to set *Pass* verdicts to states of the product whose component in $\mathcal{TP}$ is accepting. Finally, the analysis of co-reachability from *Pass* states allows both to set *None* verdicts, and *Inc* ones by complementation, and finally to select a test case $\mathcal{TC}$ by removing those transitions labelled by outputs ending in *Inc* and all transitions from *Inc*. Here, the aim is essentially to mimic this computation process for the case of RTSs. This means that computations are here 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 any visible reaction in the specification. Given a specification defined by an RTS $\mathcal{R}$, detecting vertices where quiescence is permitted enables to construct a suspended specification, $\Delta(\mathcal{R})$.

For finite state IOLTSs, livelocks come from loops. For infinite state IOLTSs (*e.g.* defined by RTSs), livelocks may also come from infinite paths of silent actions involving infinitely many states. Such paths are said *divergent*. The following lemma characterizes the existence of loops or divergent paths in RTSs by the existence of a path between two copies of the same vertex.

Lemma 1. *For an 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 $q_1 \xrightarrow{w} q_2$ for some $w \in (\Sigma^\tau)^+$ and for all states q on this path, $\ell(q_1) \leq \ell(q)$.*

Proof. ($\Rightarrow$) Let $p = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} q_2 ...$ be an infinite path in $[\![\mathcal{R}]\!]$, with $\forall k \in \mathbb{N}, \mu_k \in \Sigma^\tau$. If p contains a loop, there exits one state of minimal level in this loop, let q_{i_1} be this state. Now consider an elementary path (*i.e.* a path with no loop). As each state in this path 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 that two states of $[\![v]\!]$ appear in this path. Let these two states be q_1 and q_2.

($\Leftarrow$) Suppose that there exists a vertex v and two states $q_1, q_2 \in [\![v]\!]$ such that $q_1 \xrightarrow{w} q_2$ for $w \in (\Sigma^\tau)^+$, and for all states q on this path, $\ell(q_1) \leq \ell(q)$. There are two cases. If $\ell(q_1) = \ell(q_2)$ 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 be a path $q_1 \xrightarrow{w} q_2$ for $w \in (\Sigma^\tau)^+$, such that for all q in this path, $\ell(q_1) \leq \ell(q)$. Thus, from the definition of tiling, a similar path, p_1, may be constructed from q_2 to a state $q_3 \in [\![v]\!]$, with, $p_1 = q_2 \xrightarrow{\sigma'} q_3$ 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 of silent actions in $[\![\mathcal{R}]\!]$: a divergent path. □

The next proposition states that it is effective to build the suspension of an RTS, *i.e.* , an RTS whose semantics is the suspension (in terms of IOLTS) of the semantics of the original RTS.

Proposition 7. *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* $\mathsf{Traces}([\![\Delta(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!])$.

Proof. Let $\mathcal{R}$ be a RTS, self-loops labelled by δ are added as follows.

- For output quiescence (deadlock or absence of output), Requirement 1, item 2 (defined after Definition 9), ensures that $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!])$, for a vertex v in a tile t of $\mathcal{R}$, has a uniform value. The δ-transitions can thus be added to each v in $\mathcal{R}$ such that $\Gamma_{[\![\mathcal{R}]\!]}([\![v]\!]) \subseteq \Sigma^?_{\mathcal{R}}$. This operation produces a new RTS $\mathcal{R}'$.
- For livelocks, the two different cases of internal loops and *divergent paths* are tackled by Lemma 1. Such situations may be detected from self-reaching vertices. This result also ensures that this detection may be performed by considering each tile as an axiom. Then, for each tile t in $\mathcal{R}'$, the following is performed:
  - Each vertex v of tile t is coloured by a new colour λ_v not in $\Lambda_{\mathcal{R}'}$.
  - Proposition 2 is used 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. Quiescence is added to each such vertex in $\mathcal{R}'$ to produce $\Delta(\mathcal{R})$.

It is not hard to see that this construction mimics the suspension of IOLTSs on RTSs, thus ensuring that $\mathsf{Traces}([\![\Delta(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!])$. □

Output completion

After using Proposition 7 for the computation of $\Delta(\mathcal{R})$ from the specification $\mathcal{R}$, the next step is to complete $\Delta(\mathcal{R})$ into an RTS denoted $CS(\mathcal{R})$ which recognize $\mathsf{STraces}(\mathcal{R}).\Sigma^{!\delta}$ in a fresh color.

The complete suspended specification, denoted by $CS(\mathcal{R})$, is computed from $\Delta(\mathcal{R})$ as follows: a fresh colour *UnS* is added to detect paths ending with unspecified outputs. Then, for every tile $t_{\mathcal{A}}$, a new vertex $v_{\mathcal{A}}^{UnS}$ is added (having colour *UnS*), and new transitions leading to $v_{\mathcal{A}}^{UnS}$ are added as well for unspecified outputs:

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

Remember that $\Gamma_{[\![\Delta(\mathcal{R})]\!]}([\![v]\!])$ is uniform, by Requirement 1, item 2.

By construction, the following is true:

$$\mathsf{Traces}_{\mathcal{C}(UnS)}([\![CS(\mathcal{R})]\!]) \subseteq \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \quad (1)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![CS(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (2)$$

$$\mathsf{Traces}([\![CS(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \cup \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (3)$$

The inequality (1) simply says that the traces of sequences recognized in *UnS* are suspension traces of $\mathcal{R}$ prolonged with outputs. The equality (2) holds because $\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![CS(\mathcal{R})]\!])$, which are traces of sequences leading outside the colour *UnS*, are the original suspension traces of $\Delta(\mathcal{R})$, thus $\mathsf{STraces}([\![\mathcal{R}]\!])$. The equality (3) is obtained by union of (1) and (2). Notice however that it is not a disjoint union: a trace can be in both $\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![CS(\mathcal{R})]\!])$ and $\mathsf{Traces}_{\mathcal{C}(UnS)}([\![CS(\mathcal{R})]\!])$, as it can be the projection of both a sequence in $\Delta(S)$ and a sequence leading to *UnS*.

Expanding the definition of $\mathsf{MinFTraces}([\![\mathcal{R}]\!])$, produces

$$\mathsf{MinFTraces}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}(UnS)}([\![CS(\mathcal{R})]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![CS(\mathcal{R})]\!]) \quad (4)$$

Remark 3. *Notice that the introduction of the section describes a test generation process from finite IOLTSs, where a canonical tester is built by output completion to Fail after determinization. A similar process could be described for weighted RTSs. However, here, the process consists first in performing an output completion (with a slightly different meaning to a colour UnS), and then in determinizing (next paragraph) while defining Fail. The reason is that output completion can be computed for any RTS, and will be used for both off-line and on-line test generation, while determinization is not, and will be used only for off-line test generation for weighted RTSs.*

Σ^τ-closure

Using Proposition 4, from $CS(\mathcal{R})$ one can build an RTS $Clo(CS(\mathcal{R}))$, which semantics $[\![Clo(CS(\mathcal{R}))]\!]$ has no internal action and has same coloured traces as $[\![CS(\mathcal{R})]\!]$. It immediately follows that $Clo(CS(\mathcal{R}))$ ensures the same inequalities and equalities as (1), (2) and (3):

$$\mathsf{Traces}_{\mathcal{C}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) \subseteq \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \quad (5)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (6)$$

$$\mathsf{Traces}([\![Clo(CS(\mathcal{R}))]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \cup \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (7)$$

Moreover, the equality (4) immediately transposes to $Clo(CS(\mathcal{R}))$:

$$\mathsf{MinFTraces}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) \quad (8)$$

Canonical tester

Whenever $Clo(CS(\mathcal{R}))$ is weighted, Proposition 6 enables to determinize it into $\mathcal{D}(Clo(CS(\mathcal{R})))$. From $\mathcal{D}(Clo(CS(\mathcal{R})))$ a new RTS $Can(\mathcal{R})$ called the *canonical tester* of $\mathcal{R}$ is built as follows:

- a new colour *Fail* is considered and vertices of $\mathcal{D}(Clo(CS(\mathcal{R})))$ are coloured by *Fail* if composed of vertices all coloured by *UnS* in $Clo(CS(\mathcal{R}))$, thus recognizing traces of sequences all leading to *UnS*.
- inputs and outputs are mirrored in $Can(\mathcal{R})$ wrt. $\mathcal{R}$.

From this construction and equality (8) follows:

$$\mathsf{Traces}_{\mathcal{C}(\mathit{Fail})}([\![Can(\mathcal{R})]\!]) = \mathsf{MinFTraces}([\![\mathcal{R}]\!]) \quad (9)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{Fail})}([\![Can(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (10)$$

and

$$\mathsf{Traces}([\![Can(\mathcal{R})]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \cup \mathsf{MinFTraces}([\![\mathcal{R}]\!]) \quad (11)$$

where the union is now a disjoint union.

In fact

$$\mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{Fail})}([\![Can(\mathcal{R})]\!]) = \mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \text{ by equality (6)}$$

and

$$\begin{aligned}\mathsf{Traces}_{\mathcal{C}(\mathit{Fail})}([\![Can(\mathcal{R})]\!]) &= \mathsf{Traces}_{\mathcal{C}(\mathit{UnS})}([\![Clo(CS(\mathcal{R}))]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(\mathit{UnS})}([\![CS(\mathcal{R})]\!]) \\ &= \mathsf{MinFTraces}([\![\mathcal{R}]\!]) \text{ by equality (8)}\end{aligned}$$

From equality (9) it immediately follows that the test suite $\mathcal{TS}$ reduced to the canonical tester, $\mathcal{TS} = \{Can(\mathcal{R})\}$, is sound and exhaustive (see Section 2). $\mathcal{TS}$ is also strict, which is proved as follows:

$$\mathsf{Traces}([\![Can(\mathcal{R})]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}(\mathit{Fail})}([\![Can(\mathcal{R})]\!])$$

using the fact that (11) is a disjoint union, and the equality (10).

Example 4. *Figure 4 represents the canonical tester obtained from Example 1. Observe that it has been suspended, and that internal actions,* $\{\texttt{try}, \texttt{throw}, \texttt{catch}, \texttt{intern}\}$*, have been abstracted away. The vertices labelled by F correspond to the ones coloured by Fail. These vertices are those reached by unspecified output actions.*

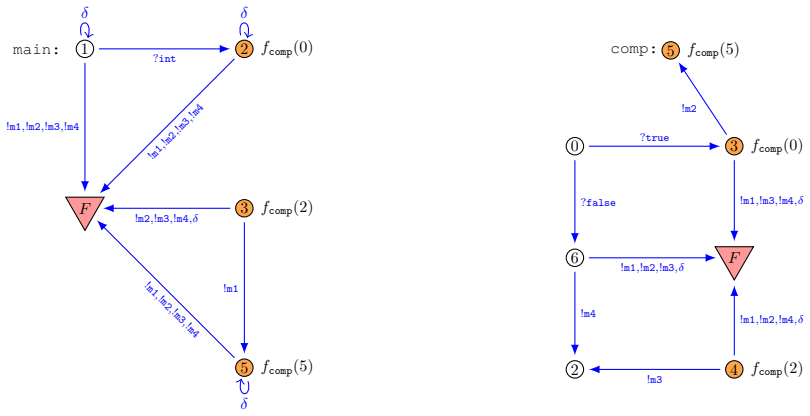

Figure 4: Example of a canonical tester.

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 the present formal framework, a test purpose will be defined by a deterministic IOLTS, using the fact that the product of an RTS and an IOLTS is still an RTS.

Definition 12 (Test purpose)**.** *A* test purpose *is a complete deterministic finite IOLTS $\mathcal{TP}$ over $\Sigma^{\circ\delta}$, with a particular colour Accept, such that states coloured by Accept have no successors.*

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

- $\mathcal{C}_{\mathcal{P}}(\mathit{Fail}) = \mathcal{C}_{Can(\mathcal{R})}(\mathit{Fail}) \times Q_{\mathcal{TP}}$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Pass}) = \overline{\mathcal{C}}_{\mathcal{P}}(\mathit{Fail}) \times \mathcal{C}_{\mathcal{TP}}(\mathit{Accept})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{None}) = Coreach(\Sigma^{\circ\delta}, \mathcal{C}_{\mathcal{P}}(\mathit{Pass})) \setminus \mathcal{C}_{\mathcal{P}}(\mathit{Pass})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Inc}) = Q_P \setminus (\mathcal{C}_{\mathcal{P}}(\mathit{Fail}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{Pass}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{None}))$

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

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

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

Example 5. *Using the canonical tester (Fig. 4) resulting from Example 1, Figure 5 depicts the test case obtained with the test purpose, $\mathcal{TP}$, accepting the traces in $(\Sigma^{\circ\delta})^*.$*`?true`*.`*`?true`*$(\Sigma^{\circ\delta})^*.$*`!m1`*. The IOLTS $\mathcal{TP}$ has four states $q_1$, $q_2$, $q_3$ and $q_4$ (the only state coloured Accept), with self-loops for all actions but one in $q_1$ and $q_3$, and transitions labelled respectively by* `?true` *and* `!m1`*, from q_1 to q_2 and from q_3 to q_4.*

The product between the canonical tester and $\mathcal{TP}$ is performed in each tile leading to the two tiles depicted in Figure 5. For a better readability, the only vertices represented are those reachable from $(1, q_1)$ (of tile `main`*) furthermore, in each tile, vertices coloured by Fail (*resp.*, Inc) are merged.*

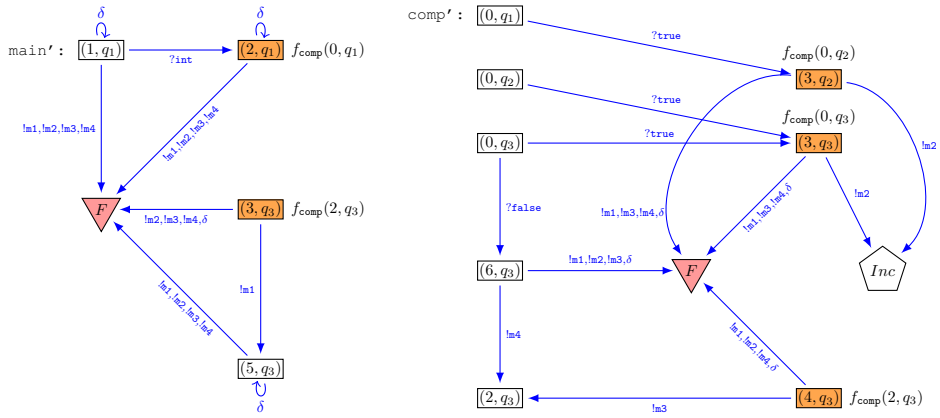

Figure 5: Example of a test case

4.2 Complexity of the computation of the canonical tester

The computation of the canonical tester is the core of formal test generation. In the case of RTSs it follows several steps described in the previous subsection.

There are two exponential steps in the computation of the canonical tester: the determinization which is unavoidable, and the Σ^τ-closure (computation of $Clo(CS(\mathcal{R}))$). The latter results from the possibility to have traversing Σ^τ-paths, in fact, complete trees of Σ^τ-transitions are needed to reach this bound. The following list summerize the complexity of each individual step.

1) Computation of $\Delta(\mathcal{R})$: **polynomial time**

2) Computation of $CS(\mathcal{R})$: **polynomial time**

3) Computation of $Clo(CS(\mathcal{R}))$: **exponential space**

4) Checking whether $Clo(CS(\mathcal{R}))$ is weighted: **polynomial time**

5) Determinizing $Clo(CS(\mathcal{R}))$: **exponential space**

6) Synchronous product $Can(\mathcal{R}) \times \mathcal{TP}$ and selection: **polynomial time**

4.3 Properties of generated test cases

This subsection contains proofs of 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 $\mathcal{P} = Can(\mathcal{R}) \times \mathcal{TP}$, the definition of $\mathcal{C}_\mathcal{P}(Fail)$, and pruning, selection by $\mathcal{TP}$ does not add any colouring by *Fail* with respect to $Can(\mathcal{R})$, thus $\mathsf{Traces}_{\mathcal{C}(Fail)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}([\![\mathcal{TC}]\!]) \cap \mathsf{Traces}_{\mathcal{C}(Fail)}([\![Can(\mathcal{R})]\!])$. The equation $\mathsf{Traces}_{\mathcal{C}(Fail)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}([\![\mathcal{TC}]\!]) \cap \mathsf{MinFTraces}([\![\mathcal{R}]\!]) \subseteq \mathsf{MinFTraces}([\![\mathcal{R}]\!])$ follows from equation (9) and proves both strictness (equality) and soundness (inclusion).

Exhaustiveness

It is now possible to prove that the test suite $\mathcal{TS}$ composed of all test cases that can be generated from arbitrary test purposes $\mathcal{TP}$ is exhaustive. First one needs to establish the inequality $\bigcup_{\mathcal{TC}\in\mathcal{TS}} \mathsf{Traces}_{\mathcal{C}(Fail)}([\![\mathcal{TC}]\!]) \supseteq \mathsf{MinFTraces}([\![\mathcal{R}]\!])$.

Let $\sigma' = \sigma.a \in \mathsf{MinFTraces}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}(Fail)}([\![Can(S)]\!])$ be a minimal non-conformant trace for $\mathcal{R}$. Thus σ belongs to $\mathsf{STraces}([\![\mathcal{R}]\!])$ and there exists $b \in \Sigma^{!\delta}$ such that $\sigma.b \in \mathsf{STraces}([\![\mathcal{R}]\!])$ (if no output continues σ in $\mathsf{STraces}([\![\mathcal{R}]\!])$, a δ does). Now, it remains to define a test purpose $\mathcal{TP}$ such that $\sigma.b \subseteq \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP})$. Let $\mathcal{TC}$ be the test case generated from $\mathcal{R}$ and $\mathcal{TP}$. By construction of $\mathcal{TC}$ from $\mathcal{R}$ and $\mathcal{TP}$: both $\sigma.b$ belongs to $\mathsf{Traces}([\![\mathcal{TC}]\!])$ and σ' belongs to $\mathsf{Traces}_{\mathcal{C}(Fail)}([\![\mathcal{TC}]\!])$. The requested inclusion is thus established.

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. This may be formalized as follows:

Definition 13 (Precision)**.** *A test case $\mathcal{TC}$ is* precise *with respect to an IOLTS specification $\mathcal{M}$ and a test purpose $\mathcal{TP}$ if* $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}(\mathcal{M}) \cap \mathsf{Traces}([\![\mathcal{TC}]\!])$.

It is easy to prove that test cases generated from an RTS $\mathcal{R}$ and a test purpose $\mathcal{TP}$ are precise. By construction, states coloured by *Pass* are those coloured by *Accept* in $\mathcal{TP}$ and not by *Fail* in $Can(\mathcal{R})$. Thus $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) = \mathsf{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \mathsf{STraces}([\![\mathcal{R}]\!])$, which (since $\mathsf{Traces}_{\mathcal{C}(Pass)}([\![\mathcal{TC}]\!]) \subseteq \mathsf{Traces}([\![\mathcal{TC}]\!])$) implies precision.

5 On-line test generation from RTS

Like every model characterizing context-free languages, RTSs are not determinizable (as seen in Section 3). This issue does not doom the prospect of formal test suites generation. In similar cases, Tretmans [1] suggests an on-line test generation process. In fact this process amounts to producing test cases without constructing a deterministic canonical tester. Such a technique performed either off-line or on-line is applicable to RTSs. This section introduces this technique, and establishes properties of the generated test cases.

5.1 Test case generation

Since the only transformation not guaranteed to succeed in off-line test generation is determinization, the first steps of the algorithm to generate test cases on-line are identical to those generating test cases off-line, namely suspension, output-completion and Σ^τ-closure. The process thus starts from the closure of the output-completed specification $Clo(CS(\mathcal{R}))$ defined in Section 4. This time, the canonical tester cannot be built in general by determinization from $Clo(CS(\mathcal{R}))$. However, using Proposition 4, one can build $Clo(CS(\mathcal{R}))$ from $\mathcal{R}$, ensuring the following properties of equations (5), (6), (7) and (8):

$$\mathsf{Traces}_{\mathcal{C}(UnS)}([\![Clo(CS(\mathcal{R}))]\!]) \subseteq \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \quad (5)$$

$$\mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![Clo(CS(\mathcal{R}))]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (6)$$

$$\mathsf{Traces}([\![Clo(CS(\mathcal{R}))]\!]) = \mathsf{STraces}([\![\mathcal{R}]\!]).\Sigma^{!\delta}_{\mathcal{R}} \cup \mathsf{STraces}([\![\mathcal{R}]\!]) \quad (7)$$

and

$$\mathsf{MinFTraces}([\![\mathcal{R}]\!]) = \mathsf{Traces}_{\mathcal{C}(UnS)}([\![Clo(CS(\mathcal{R}))]\!]) \setminus \mathsf{Traces}_{\overline{\mathcal{C}}(UnS)}([\![Clo(CS(\mathcal{R}))]\!]) \quad (8)$$

5.1.1 Product and colouring

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

- $\mathcal{C}_{\mathcal{P}}(\mathit{UnS}) = \mathcal{C}_{Clo(CS(\mathcal{R}))}(\mathit{UnS}) \times Q_{\mathcal{T}P}$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Pass}) = \overline{\mathcal{C}}_{Clo(CS(\mathcal{R}))}(\mathit{UnS}) \times \mathcal{C}_{\mathcal{T}P}(\mathit{Accept})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{None}) = Coreach(\Sigma^{o\delta}, \mathcal{C}_{\mathcal{P}}(\mathit{Pass})) \setminus \mathcal{C}_{\mathcal{P}}(\mathit{Pass})$
- $\mathcal{C}_{\mathcal{P}}(\mathit{Inc}) = Q_{\mathcal{P}} \setminus (\mathcal{C}_{\mathcal{P}}(\mathit{Fail}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{Pass}) \cup \mathcal{C}_{\mathcal{P}}(\mathit{None}))$

5.1.2 Computing test cases

The last step consists in computing test cases in the form of IOLTSs, by exploration of the semantics of $Clo(CS(\mathcal{R}))$, in a similar way as Tretmans [1]. These test cases will be modelled as finite trees formed by alternating sequences of choices of inputs for the system and subtrees of possible answers of the system (computed from $\mathcal{P}$), each node of the tree carries a verdict.

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, and some symbol a, the tree formed by a followed by tree θ is denoted by $a;\theta$, it is defined by $a;\theta \triangleq \{au \mid u \in \theta\}$. 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 of $[\![cloCS(\mathcal{R})]\!]$. Test cases are defined by applying the following algorithm recursively, starting from the initial state $\mathcal{C}_{\mathcal{P}}(init)$.

On-line test generation algorithm.

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 recursively 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 recursively 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. It is skipped for readability.

At every step, the algorithm makes a non-deterministic choice: namely to stop or to proceed. This choice might be influenced by several factors, for example, in order to avoid generating tests cases containing neither *Fail* nor *Pass*.

5.2 Properties of the test cases generated on-line

One of the main benefits of the formal generation of test cases from a specification is that properties of these test suites may be proved. This is still the case for on-line test suites. Even though these proofs are largely similar to those of the off-line case, we present precise correlations in this section.

Soundness and Strictness

By definition of the set X_1, the traces of $\mathcal{TC}$ falling in a state coloured by *Fail* are those in $\mathsf{Traces}(\llbracket Clo(CS(\mathcal{R}))\rrbracket) \setminus \mathsf{Traces}_{\overline{\mathcal{C}(_{UnS})}}(\llbracket Clo(CS(\mathcal{R}))\rrbracket) = \mathsf{MinFTraces}(\llbracket\mathcal{R}\rrbracket)$. Thus $\mathsf{Traces}_{\mathcal{C}(_{Fail})}(\mathcal{TC}) = \mathsf{MinFTraces}(\llbracket\mathcal{R}\rrbracket) \cap \mathsf{Traces}(\mathcal{TC})$ 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 $\mathsf{Traces}_{\mathcal{C}(_{Pass})}(\mathcal{TC}) = \mathsf{Traces}_{\mathcal{C}(_{Pass})}(Clo(CS(\mathcal{R})) \times \mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$. Then, by definitions of the colours: $\mathsf{Traces}_{\mathcal{C}(_{Pass})}(\mathcal{TC}) = \mathsf{Traces}_{\overline{\mathcal{C}}_{UnS}}(Clo(CS(\mathcal{R}))) \cap \mathsf{Traces}_{\mathcal{C}(_{Accept})}(\mathcal{TP})) \cap \mathsf{Traces}(\mathcal{TC})$. Which eventually proves precision: $\mathsf{Traces}_{\mathcal{C}(_{Pass})}(\mathcal{TC}) = \mathsf{STraces}(\mathcal{R}) \cap \mathsf{Traces}_{\mathcal{C}(_{Accept})}(\mathcal{TP}) \cap \mathsf{Traces}(\mathcal{TC})$.

5.3 Application of on-line test cases generation

Previous subsections have shown how to generate on-line test cases without computing a deterministic canonical tester. In fact, using the method proposed in Section 3.4, it is possible to perform such computation without constructing the whole tile modeling the system. From the algorithm presented in subsection 5.1.2, we compute a set of compatible symbolic paths: each of these paths is stored as a pair formed by a vertex and a word of tile symbols (in T^*).

The following example illustrates how to apply the previous construction to the RTS of Example 1 (it is deterministic but is convenient for the purpose of illustration).

Example 6. *Let T be the set $\{m, c\}$ where the symbols represent respectively tiles* `main` *and* `comp`*. Assume $8, true, true, true$ is the sequence of input already computed. One may check that the following pair is reached: $(1, mccc)$. Hence if the implementation outputs something, the test will* **fail**. *Otherwise if, for example, the random choice of the tester selects to output $false$, $(6, mcccc)$ is reached, then the output* `"You stopped"` *(message* `m4`*) is expected, leading to the configuration $(2, mcccc)$. Then the tester only expects the message* `"Some text"` *(message* `m3`*), each time removing one c. Eventually, state $(5, m)$ will be reached after receiving* `"Done"` *(message* `m1`*) from the implementation. This would be the end of this test.*

6 Conclusion

This paper presented an account on recursive tile systems, a general model of IOLTSs allowing for recursion. It 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 key properties. The precision of these tests with respect to test purposes has been established. Moreover precise assessments of the complexities of involved operations have been provided.

This method has a drawback: the classical off-line approach may not be used whenever the RTS is not weighted. This property may be verified in polynomial time, but it would really be comforting to have a syntactical characterization of a class of RTSs being weighted. Identifying such a class would be a natural continuation of this work.

Another interesting perspective would be to incorporate known results on probabilistic RTSs that have been considered by several authors [19, 20]. This would enable to take into account quantitative properties of systems, or to express coverage properties of finite test suites.

References

[1] Tretmans J. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools* 1996; **17**(3):103–120.

[2] Jard C, Jéron T. TGV: theory, principles and algorithms. *Software Tools for Technology Transfer (STTT)* 2005; **7**(4):297–315.

[3] Tretmans J, Brinksma H. Torx: Automated model-based testing. *First European Conference on Model-Driven Software Engineering*, Hartman A, Dussa-Ziegler K (eds.), 2003; 31–43. URL `http://doc.utwente.nl/66990/`.

[4] Jeannet B, Jéron T, Rusu V. Model-based test selection for infinite-state reactive systems. *5th International Symposium on Formal Methods for Components and Objects (FMCO'06), Revised Lectures*, *LNCS*, vol. 4709, 2006; 47–69.

[5] Frantzen L, Tretmans J, Willemse T. A symbolic framework for model-based testing. *FATES 2006 and RV 2006, Revised Selected Papers*, *LNCS*, vol. 4262, 2006; 40–54.

[6] Larsen K, Mikucionis M, Nielsen B. Online testing of real-time systems using Uppaal. *Formal Approaches to Software Testing (FATES'04)*, *LNCS*, vol. 3395, 2005; 79–94.

[7] Krichen M, Tripakis S. Conformance testing for real-time systems. *Formal Methods in System Design* 2009; **34**(3):238–304.

[8] Bertrand N, Jéron T, Stainer A, Krichen M. Off-line test selection with test purposes for non-deterministic timed automata. *Logical Methods in Computer Science* 2012; **8**(4).

[9] Alur R, Etessami K, Yannakakis M. Analysis of recursive state machines. *13th International Conference on Computer Aided Verification, (CAV'01)*, *LNCS*, vol. 2102, 2001; 207–220.

[10] Courcelle B. *Handbook of Theoretical Computer Science*, chap. Graph rewriting: an algebraic and logic approach. Elsevier, 1990.

[11] Caucal D. Deterministic graph grammars. *Logic and Automata*, *Texts in Logic and Games*, vol. 2, Flum J, Grädel E, Wilke T (eds.), Amsterdam University Press, 2008; 169–250.

[12] Caucal D, Hassen S. Synchronization of grammars. *Third International Computer Science Symposium in Russia (CSR'08)*, *LNCS*, vol. 5010, 2008; 110–121.

[13] Chédor S, Morvan C, Pinchinat S, Marchand H. Analysis of partially observed recursive tile systems. *11th edition of Workshop on Discrete Event Systems*, Guadalajara, Mexico, 2012; 265–271.

[14] Constant C, Jeannet B, Jéron T. Automatic test generation from interprocedural specifications. *TestCom/FATES'07*, *LNCS*, vol. 4581, 2007; 41–57.

[15] Hassen S. Synchronisation de grammaires de graphes. Phd thesis, Université de la Réunion 2008.

[16] Alur R, Madhusudan P. Visibly pushdown languages. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*, ACM, 2004; 202–211.

[17] Caucal D. Synchronization of regular automata. *34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, *LNCS*, vol. 5734, 2009; 2–23.

[18] Nowotka D, Srba J. Height-deterministic pushdown automata. *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*, *LNCS*, vol. 4708, 2007; 125–134.

[19] Esparza J, Kučera A, Mayr R. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science* 2006; **2**(1).

[20] Bertrand N, Morvan C. Probabilistic regular graphs. *12th International Workshop on Verification of Infinite-State Systems, INFINITY'10*, *EPTCS*, vol. 39, Singapour, 2010; 77–90.