



# Test Generation from Recursive Tile Systems

Sébastien Chédor, Thierry Jéron, Christophe Morvan

► **To cite this version:**

| Sébastien Chédor, Thierry Jéron, Christophe Morvan. Test Generation from Recursive Tile Systems. [Research Report] RR-8206, INRIA. 2013, pp.32. <hal-00778134>

**HAL Id: hal-00778134**

**<https://hal.inria.fr/hal-00778134>**

Submitted on 18 Jan 2013

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



# Test Generation from Recursive Tile Systems

Sébastien Chédor, Thierry Jéron, Christophe Morvan

**RESEARCH  
REPORT**

**N° 8206**

Janvier 2013

Project-Teams VerTeCs

ISRN INRIA/RR--8206--FR+ENG

ISSN 0249-6399





## Test Generation from Recursive Tile Systems

Sébastien Chédor\*, Thierry Jéron†, Christophe Morvan‡

Project-Teams VerTeCs

Research Report n° 8206 — Janvier 2013 — 29 pages

**Abstract:** In this paper we explore 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 models is seldom explored in the literature. We first present an off-line test generation algorithm for *Weighted* RTSs, a determinizable sub-class of RTSs, and second, an on-line test generation algorithm for the full RTS model. Both algorithms use test purposes to guide test selection through targeted behaviours. Additionally, essential properties relating verdicts produced by generated test cases on an implementation with both the conformance with respect to its specification, and the precision with respect to a test purpose, are proved.

**Key-words:** Model-based Testing, IOCO Theory, Recursive Systems

---

\* Université de Rennes I, Campus de Beaulieu, 35042 Rennes, France

† INRIA Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes, France

‡ Université Paris-Est, Marne-La-Vallée, France

**RESEARCH CENTRE  
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu  
35042 Rennes Cedex

## **Génération de tests pour les systèmes de tuile récurifs**

**Résumé :** Dans ce rapport, nous nous intéressons à la génération de cas de tests de conformité pour les systèmes de tuiles récurifs (RTSs) dans le contexte classique de la théorie **ioco**. Le modèle des RTSs permet de décrire les systèmes réactifs avec un comportement récurif, et est très similaire aux automates à pile, aux grammaires de graphes avec remplacement d'hyperarcs et aux machines récurives à états. La génération de test pour ce genre de modèles est rarement traitée dans la littérature. Dans un premier temps, nous présentons un algorithme de génération de test hors ligne pour les RTSs pondérés, une sous-classe de RTSs, puis, un algorithme en ligne pour le modèle complet. Les deux algorithmes utilisent des objectifs de test pour guider le test vers des comportements ciblés. De plus, les verdicts renvoyés par l'exécution des cas de tests sur une implémentation sont prouvés à la fois conformes par rapport à la spécification et précis par rapport à l'objectif de test.

**Mots-clés :** Test basé sur les modèles, Théorie IOCO, Systèmes Récurifs

## 1 Introduction and motivation

Conformance testing is the problem of checking by test experiments that a black-box implementation behaves correctly with respect to its specification. It is well known that testing is the most used validation technique to assess the quality of software systems, and represents the largest part in the cost of software development. Automatising is thus required in order to improve the cost and quality of the testing process. In particular, it is undoubtedly interesting to automate the test generation phase from specifications of the system. Formal model-based testing aims at resolving this problem by the formal description of testing artefacts (specifications, possible implementations, test cases) 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 [20] is 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 [14, 19] and for more general models whose semantics can be expressed in the form of infinite state IOLTSs [15, 12]. Test generation techniques have also been devised for timed automata models whose semantics are infinite state systems [17, 16, 4].

One can distinguish two different approaches in test generation: *off-line* test generation aims at generating test cases, store 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.

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 be able to prove properties relative to the generated tests. There are several models between finite state and Turing powerful systems; in this paper we consider 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 our running example which represents an abstraction of the one in Figure 1.

There exist several ways to define recursive behaviours: PDAs, recursive state machines [1], regulars graphs, defined by functional (or deterministic) hyperedge replacement grammars (HR-grammars), [10, 5]. 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 define classes of such systems which may be determinized [7], which is of interest for test generation. The HR-grammars, on the other hand, are very technical to define. Here we try to get the best of both worlds: we use HR-grammars presented as tiling systems, called Recursive Tile Systems (RTSs for short). These systems have already been used in the context of diagnosis [8]; 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 partitionned 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 of our knowledge test generation for recursive programs has been seldom considered in the literature. The only work we are aware of is [9] which considers a model of deterministic PDA with

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

Figure 1: A recursive program

inputs/outputs (IOPDS) and generate test cases in the same model. The present work can be seen as an extension of this, where non-determinism is taken into account.

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

## 2 Conformance testing theory for IOLTSSs

This section recalls the **io** testing theory introduced by Tretmans [20] for the model of Input/Output Labelled Transition Systems (IOLTSSs) that will serve as a basis for test generation from RTSs. We first give a non-standard definition of IOLTSSs where marking of states is defined by colors, and introduce notations and basic operations on IOLTSSs. We then review the **io** testing theory, with the modelling of test artifacts and their interactions, the central notion of conformance relation, and essential properties requested on test cases.

### 2.1 The IOLTSS model and operations

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

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

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

The language of  $\mathcal{M}$  accepted in a set of states  $P \subseteq Q_{\mathcal{M}}$  noted  $L_P(\mathcal{M}) \triangleq \{w \in (\Sigma_{\mathcal{M}})^* \mid \exists q_0 \in \mathcal{C}(\text{init}_{\mathcal{M}}), q \in P : q_0 \xrightarrow{w}_{\mathcal{M}} q\}$ , is the set of sequences from an initial state to a state in  $P$ . In particular  $L(\mathcal{M}) \triangleq L_{Q_{\mathcal{M}}}(\mathcal{M})$  represents the whole set of sequences of  $\mathcal{M}$ . We say that a sequence is accepted in a colour  $\lambda$  if it is accepted in  $\mathcal{C}(\lambda)$  and note  $L_{\lambda}(\mathcal{M})$  for  $L_{\mathcal{C}(\lambda)}(\mathcal{M})$ .

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

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

Visible behaviours of  $\mathcal{M}$ , which are essential to consider for testing, are defined as usual by the relation  $\Longrightarrow_{\mathcal{M}} \in Q_{\mathcal{M}} \times (\{\epsilon\} \cup \Sigma_{\mathcal{M}}^{\circ}) \times Q_{\mathcal{M}}$  as follows:  $q \xRightarrow{\epsilon}_{\mathcal{M}} q' \triangleq q = q'$  or  $q \xrightarrow{\tau_1 \cdot \tau_2 \cdots \tau_n}_{\mathcal{M}} q'$ , for  $\tau_i \in \Sigma_{\mathcal{M}}^{\tau}$  and for  $a \in \Sigma_{\mathcal{M}}^{\circ}$ ,  $q \xRightarrow{a}_{\mathcal{M}} q' \triangleq \exists q_1, q_2 : q \xRightarrow{\epsilon}_{\mathcal{M}} q_1 \xrightarrow{a}_{\mathcal{M}} q_2 \xRightarrow{\epsilon}_{\mathcal{M}} q'$ . The notation is extended to sequences as follows: for  $\sigma = a_1 \cdots a_n \in (\Sigma_{\mathcal{M}}^{\circ})^*$  a sequence of visible actions,  $q \xRightarrow{\sigma}_{\mathcal{M}} q'$  stands for  $\exists q_0, \dots, q_n : q = q_0 \xRightarrow{a_1}_{\mathcal{M}} q_1 \cdots \xRightarrow{a_n}_{\mathcal{M}} q_n = q'$  and  $q \xRightarrow{\sigma}_{\mathcal{M}}$  for  $\exists q' : q \xRightarrow{\sigma}_{\mathcal{M}} q'$ .

<sup>1</sup> In the examples, for readability reasons, we write  $?a$  for an input  $a \in \Sigma_{\mathcal{M}}^?$ ,  $!x$  for an output  $x \in \Sigma_{\mathcal{M}}^!$  and internal actions have no sign.



We denote  $q$  after  $\sigma \triangleq \{q' \in Q \mid q \xrightarrow{\sigma}_{\mathcal{M}} q'\}$  for the set of states in which  $\mathcal{M}$  can be after observing the visible sequence  $\sigma$  starting from the state  $q$ . The notation is extended to sets of states: for  $P \subseteq Q_{\mathcal{M}}$ ,  $P$  after  $\sigma \triangleq \bigcup_{q \in P} q$  after  $\sigma$ .

For a state  $q$ ,  $\text{Traces}(q) \triangleq \{\sigma \in (\Sigma_{\mathcal{M}}^{\circ})^* \mid q \xrightarrow{\sigma}_{\mathcal{M}}\}$  denotes the set of sequences of visible actions (called *traces*) that may be observed from  $q$  and  $\text{Traces}(\mathcal{M}) \triangleq \bigcup_{q_0 \in \mathcal{C}(\text{init}_{\mathcal{M}})} \text{Traces}(q_0)$  are those traces from the set of initial states. For a set of states  $P$ ,  $\text{Traces}_P(\mathcal{M}) = \{\sigma \in (\Sigma_{\mathcal{M}}^{\circ})^* \mid (\mathcal{C}_{\mathcal{M}}(\text{init}_{\mathcal{M}}) \text{ after } \sigma) \cap P \neq \emptyset\}$  denotes the set of traces of sequences accepted in  $P$ .

$\mathcal{M}$  is *input-complete* if in each state all inputs are enabled, possibly after internal actions, i.e.  $\forall q \in Q_{\mathcal{M}}, \forall \mu \in \Sigma_{\mathcal{M}}^?, q \xrightarrow{\mu}_{\mathcal{M}}$ .

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

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

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

**Synchronous product of 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 IOLTS thus implements the intersection of (accepted) languages:

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

As usual, for  $P_i \subseteq Q_{\mathcal{M}_i}, i = 1, 2$  we then get  $L_{P_1 \times P_2}(\mathcal{M}_1 \times \mathcal{M}_2) = L_{P_1}(\mathcal{M}_1) \cap L_{P_2}(\mathcal{M}_2)$  and in particular  $L(\mathcal{M}_1 \times \mathcal{M}_2) = L(\mathcal{M}_1) \cap L(\mathcal{M}_2)$  for the case where  $P_i = Q_{\mathcal{M}_i}, i = 1, 2$ .

**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. We will use this operation to describe the execution of test cases on an implementation.

**Definition 3** (Parallel composition). *Let  $\mathcal{M}_i = (Q_{\mathcal{M}_i}, \Sigma_{\mathcal{M}_i}, \Lambda_{\mathcal{M}_i}, \rightarrow_{\mathcal{M}_i}, \mathcal{C}_{\mathcal{M}_i}, \text{init}_{\mathcal{M}_i})$ ,  $i = 1, 2$  be two IOLTSs with mirrored visible alphabets (i.e.  $\Sigma_{\mathcal{M}_1}^! = \Sigma_{\mathcal{M}_2}^?$  and  $\Sigma_{\mathcal{M}_1}^? = \Sigma_{\mathcal{M}_2}^!$ ). Their parallel composition is the IOLTS  $\mathcal{M}_1 \parallel \mathcal{M}_2 = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \text{init}_{\mathcal{M}})$  with  $Q_{\mathcal{M}} = Q_{\mathcal{M}_1} \times Q_{\mathcal{M}_2}$ ,  $\Sigma_{\mathcal{M}} = \Sigma_{\mathcal{M}_1}$ ,  $\Lambda_{\mathcal{M}} \triangleq \Lambda_{\mathcal{M}_1} \times \Lambda_{\mathcal{M}_2}$ , in particular  $\text{init}_{\mathcal{M}} \triangleq (\text{init}_{\mathcal{M}_1}, \text{init}_{\mathcal{M}_2})$ , for any  $(\lambda_1, \lambda_2) \in \Lambda_{\mathcal{M}}$  the colouring*

relation is defined by  $\mathcal{C}_{\mathcal{M}}((\lambda_1, \lambda_2)) \triangleq \mathcal{C}_{\mathcal{M}_1}(\lambda_1) \times \mathcal{C}_{\mathcal{M}_2}(\lambda_2)$ , and the transition relation is defined by the rules:

$$\frac{a \in \Sigma^{vis} \quad q_1 \xrightarrow{a}_{\mathcal{M}_1} q'_1 \quad q_2 \xrightarrow{a}_{\mathcal{M}_2} q'_2}{(q_1, q_2) \xrightarrow{a}_{\mathcal{M}} (q'_1, q'_2)} \quad \frac{q_1 \xrightarrow{\tau}_{\mathcal{M}_1} q'_1}{(q_1, q_2) \xrightarrow{\tau}_{\mathcal{M}} (q'_1, q_2)} \quad \frac{q_2 \xrightarrow{\tau}_{\mathcal{M}_2} q'_2}{(q_1, q_2) \xrightarrow{\tau}_{\mathcal{M}} (q_1, q'_2)}$$

Synchronization being defined on visible actions, we get, for  $P_i \subseteq Q_{\mathcal{M}_i}, i = 1, 2$   $\text{Traces}_{P_1 \times P_2}(\mathcal{M}) = \text{Traces}_{P_1}(\mathcal{M}_1) \cap \text{Traces}_{P_2}(\mathcal{M}_2)$ , and in particular  $\text{Traces}(\mathcal{M}) = \text{Traces}_{P_1}(\mathcal{M}_1) \cap \text{Traces}_{P_2}(\mathcal{M}_2)$ . Note that our 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, we assume that the behaviour of the specification is modelled by an IOLTS  $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \text{init}_{\mathcal{M}})$ . 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  $\mathcal{I} = (Q_{\mathcal{I}}, \Sigma_{\mathcal{I}}, \Lambda_{\mathcal{I}}, \rightarrow_{\mathcal{I}}, \text{init}_{\mathcal{I}})$  with  $\Sigma_{\mathcal{I}} = \Sigma_{\mathcal{I}}^? \cup \Sigma_{\mathcal{I}}^! \cup \Sigma_{\mathcal{I}}^r$  and  $\Sigma_{\mathcal{I}}^? = \Sigma_{\mathcal{M}}^?$  and  $\Sigma_{\mathcal{I}}^! = \Sigma_{\mathcal{M}}^!$ . 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, we note  $\mathcal{IMP}(\mathcal{M})$  the set of such implementations, with alphabet compatible with  $\mathcal{M}$ .

**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.*  $\Gamma(q) \subseteq \Sigma_{\mathcal{M}}^?$ , (a *deadlock* *i.e.*  $\Gamma(q) = \emptyset$  is a special case of output quiescence), and a *livelock* if an infinite sequence of internal actions is enabled, *i.e.*  $\forall n \in \mathbb{N}, \exists \sigma \in (\Sigma_{\mathcal{M}}^r)^n, q \xrightarrow{\sigma}_{\mathcal{M}}$ <sup>2</sup>. We note *quiescent*( $q$ ) if  $q$  is either an output quiescence or in a livelock. From an IOLTS  $\mathcal{M}$  one can define a new IOLTS  $\Delta(\mathcal{M})$  where quiescence is made explicit by a new output  $\delta$ :

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

Note that  $\Delta(\mathcal{M})$  might be not computable for infinite state IOLTSs. In the sequel, we note  $\Sigma_{\mathcal{M}}^{! \delta}$  for  $\Sigma_{\mathcal{M}}^! \cup \{\delta\}$  and  $\Sigma_{\mathcal{M}}^{o \delta}$  for  $\Sigma_{\mathcal{M}}^o \cup \{\delta\}$ . The traces of  $\Delta(\mathcal{M})$  denoted by  $\text{STraces}(\mathcal{M})$  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 [20], 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  $\sigma$  of  $\mathcal{M}$ , the implementation  $\mathcal{I}$  exhibits only outputs and quiescences that are specified in  $\mathcal{M}$ . Formally:

<sup>2</sup> While the original **ioco** theory restricts to non-divergent IOLTSs, we here consider both loops of internal actions and divergences, *i.e.* infinite sequences of internal actions traversing an infinite number of states.

**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.  $\Sigma_{\mathcal{M}}^? = \Sigma_{\mathcal{I}}^?$  and  $\Sigma_{\mathcal{M}}^! = \Sigma_{\mathcal{I}}^!$ ),*

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

It can be proved [15] that  $\mathcal{I} \text{ ioco } \mathcal{M}$  if and only if  $\text{STraces}(\mathcal{I}) \cap \text{MinFTraces}(\mathcal{M}) = \emptyset$ , where  $\text{MinFTraces}(\mathcal{M}) \triangleq \text{STraces}(\mathcal{M}).\Sigma_{\mathcal{M}}^! \setminus \text{STraces}(\mathcal{M})$  is the set of minimal non-conformant suspension traces, where minimality refers to the prefix ordering. Notice that the set of all non-conformant traces is then  $\text{MinFTraces}(\mathcal{M}).\Sigma_{\mathcal{M}}^*$ . 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:** We now define test cases and test suites (sets of test cases), and their expected properties with respect 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 our 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  $\mathcal{TC} = (Q_{\mathcal{TC}}, \Sigma_{\mathcal{TC}}, \Lambda_{\mathcal{TC}}, \rightarrow_{\mathcal{TC}}, \mathcal{C}_{\mathcal{TC}}, \text{init}_{\mathcal{TC}})$  where  $\text{Pass}, \text{Fail}, \text{Inc}, \text{None} \in \Lambda_{\mathcal{TC}}$  are colours characterising verdicts such that  $\mathcal{C}_{\mathcal{TC}}(\text{Pass}), \mathcal{C}_{\mathcal{TC}}(\text{Fail}), \mathcal{C}_{\mathcal{TC}}(\text{Inc})$  and  $\mathcal{C}_{\mathcal{TC}}(\text{None})$  form a partition of  $Q_{\mathcal{TC}}$  and for  $\lambda \in \{\text{Pass}, \text{Fail}\}$ ,  $\text{reach}_{\mathcal{TC}}(\Sigma, \lambda) \subseteq \mathcal{C}_{\mathcal{TC}}(\lambda)$  and  $\text{reach}_{\mathcal{TC}}(\Sigma, \text{Inc}) \subseteq \mathcal{C}_{\mathcal{TC}}(\text{Inc}) \cup \mathcal{C}_{\mathcal{TC}}(\text{Fail})$ . The alphabet is  $\Sigma_{\mathcal{TC}} = \Sigma_{\mathcal{TC}}^? \cup \Sigma_{\mathcal{TC}}^!$  where  $\Sigma_{\mathcal{TC}}^? = \Sigma_{\mathcal{M}}^!$  and  $\Sigma_{\mathcal{TC}}^! = \Sigma_{\mathcal{M}}^?$  (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  $\mathcal{TC} \parallel \Delta(\mathcal{I})$  where common actions (inputs, outputs and quiescence) are synchronized. The effect is to intersect sets of suspension traces of  $\mathcal{I}$  with traces of  $\mathcal{TC}$  ( $\text{Traces}(\mathcal{TC} \parallel \Delta(\mathcal{I})) = \text{STraces}(\mathcal{I}) \cap \text{Traces}(\mathcal{TC})$ ). 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  $\mathcal{I} \text{ fails } \mathcal{TC} \triangleq \text{STraces}(\mathcal{I}) \cap \text{Traces}_{\mathcal{C}_{\mathcal{TC}}(\text{Fail})}(\mathcal{TC}) \neq \emptyset$ . Notice that  $\mathcal{I} \text{ fails } \mathcal{TC}$  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*.

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

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

- *$\mathcal{TS}$  is sound if no test case may reject a conformant implementation:*  
 $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), (\mathcal{I} \text{ ioco } \mathcal{M} \implies \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{I} \text{ fails } \mathcal{TC})).$
- *$\mathcal{TS}$  is exhaustive if it rejects all non-conformant implementations:*  
 $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), (\neg(\mathcal{I} \text{ ioco } \mathcal{M}) \implies \exists \mathcal{TC} \in \mathcal{TS}, \mathcal{I} \text{ fails } \mathcal{TC}).$
- *$\mathcal{TS}$  is complete if it is both sound and exhaustive.*
- *$\mathcal{TS}$  is strict if it detects non-conformances as soon as they happen:*  
 $\forall \mathcal{I} \in \mathcal{IMP}(\mathcal{M}), \forall \mathcal{TC} \in \mathcal{TS}, \neg(\mathcal{TC} \parallel \mathcal{I} \text{ ioco } \mathcal{M}) \implies \mathcal{I} \text{ fails } \mathcal{TC}.$

The following characterisations of soundness, exhaustiveness and strictness, derived from [15], are very convenient to prove that generated test suites satisfy those properties. They are obtained by replacing **io** 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** ([15]). *Let  $\mathcal{TS}$  be a test suite for  $\mathcal{M}$ ,*

- $\mathcal{TS}$  is sound if  $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \text{Traces}_{\mathcal{C}_{\mathcal{TC}}(\text{Fail})}(\mathcal{TC}) \subseteq \text{MinFTraces}(\mathcal{M}).\Sigma_{\mathcal{M}}^*$ ,
- $\mathcal{TS}$  is exhaustive if  $\bigcup_{\mathcal{TC} \in \mathcal{TS}} \text{Traces}_{\mathcal{C}_{\mathcal{TC}}(\text{Fail})}(\mathcal{TC}) \supseteq \text{MinFTraces}(\mathcal{M})$ ,
- $\mathcal{TS}$  is strict if  $\forall \mathcal{TC} \in \mathcal{TS}, \text{Traces}(\mathcal{TC}) \cap \text{MinFTraces}(\mathcal{M}) \subseteq \text{Traces}_{\mathcal{C}_{\mathcal{TC}}(\text{Fail})}(\mathcal{TC})$ .

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. And strictness that traces of test cases which are minimal non-conformant ones lead to a *Fail* state.

### 3 Recursive Tile Systems and their properties

In this section, we define the *Recursive Tile Systems* (RTSs), a model to define infinite state IOLTSS based on the regular graphs of [10]. We present some key properties of these systems relative to  $\varepsilon$ -closure (suppression of internal actions), product and determinization that will be useful for test generation in the next sections.

#### 3.1 Recursive tile systems

**Definition 8.** *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,
- $\Lambda$  is a finite set of colours with a particular one `init` marking initial states.
- $\mathcal{T}$  is a set of tiles  $t_A = ((\Sigma, \Lambda), Q_A, \rightarrow_A, \mathcal{C}_A, S_A, F_A)$  defined on  $(\Sigma, \Lambda)$  where
  - $Q_A \subseteq \mathbb{N}$  is the set of vertices,
  - $\rightarrow_A \subseteq Q_A \times \Sigma \times Q_A$  is a finite set of transitions,
  - $\mathcal{C}_A \subseteq Q_A \times \Lambda$  is a finite set of coloured vertices,
  - $S_A \subseteq Q_A$  is the support
  - $F_A \subseteq \mathcal{T} \times 2^{\mathbb{N} \times \mathbb{N}}$ , the frontier, relates to some tile,  $t_B$ , a partial function (often denoted  $f_B$ ) over  $\mathbb{N}$ , associating to each vertex of the support  $S_B$ , vertices of  $Q_A$ .
- $t_0 \in \mathcal{T}$  is an initial tile (the axiom), with  $S_{t_0} = \emptyset$ .

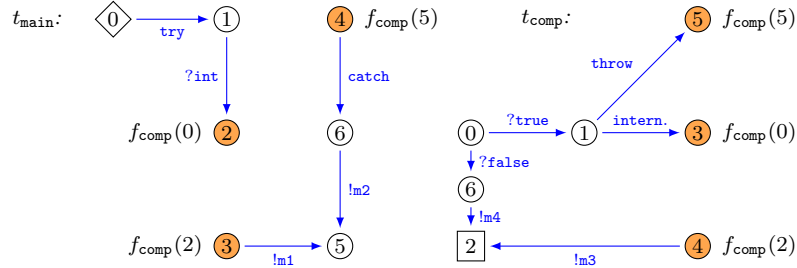
The frontier  $F_A$  of a tile  $t_A$  is used to append tiles  $t_B$  to  $t_A$ : the frontier of  $t_A$  identifies tiles  $t_B$  and how vertices of the support of  $t_B$  are merged with vertices of  $t_A$ .

For any tile  $t_A$ , we say that its  $V$ -frontier is the set of vertices which belong to the image of any function in the frontier, formally,  $V$ -frontier =  $\bigcup_{\{f_B \mid (t_B, f_B) \in F_A\}} \text{Im}(f_B)$ .

Each single tile  $t_A$  defines an IOLTS  $[t_A] = (Q_A, \Sigma, \Lambda, \rightarrow_A, \mathcal{C}_A, \text{init})$  in a straightforward way when ignoring the support and frontier.

**Example 1.** The following example presents an RTS abstracting the program of Fig. 1,  $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_{\text{main}})$  with  $\Sigma^\tau = \{\text{try, throw, catch, intern.}\}$ ,  $\Sigma^? = \{\text{int, true, false}\}$ ,  $\Sigma^! = \{\text{m1, m2, m3, m4}\}$ ,  $\Lambda = \{\text{init, succ}\}$ , a set of tiles  $\mathcal{T} = \{t_{\text{main}}, t_{\text{comp}}\}$ , and  $t_{\text{main}}$  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, observe that the actual value of this input is not reflected by the structure of the RTS, inputs `true`, `false` reflects the boolean input in block 4. The symbol `intern.` reflects the unlabelled internal action in block 5 (the computation).

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



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

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  $t_\varepsilon = ((\Sigma, \Lambda), Q_\varepsilon, \rightarrow_\varepsilon, \mathcal{C}_\varepsilon, S_\varepsilon, F_\varepsilon)$  with  $F_\varepsilon$  defined on  $\mathcal{T}$ , the tiling of  $t_\varepsilon$  by  $\mathcal{T}$ , denoted by  $\mathcal{T}(t_\varepsilon)$ , is the tile  $t'_\varepsilon = ((\Sigma, \Lambda), Q'_\varepsilon, \rightarrow'_\varepsilon, \mathcal{C}'_\varepsilon, S'_\varepsilon, F'_\varepsilon)$  iteratively defined according to the elements of the frontier  $F_\varepsilon$ , 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  $(t_B, f_B) \in F_\varepsilon$ , with  $t_B = ((\Sigma, \Lambda), Q_B, \rightarrow_B, \mathcal{C}_B, S_B, F_B) \in \mathcal{T}$ , let  $\varphi_B : Q_B \rightarrow \mathbb{N}$  be the injection mapping vertices of  $Q_B$  to new vertices of  $Q'_\varepsilon$  with  $\varphi_B(n) := f_B(n)$  whenever  $n \in \text{dom}(f_B)$ ,  $n + \max(Q'_\varepsilon) + 1$  otherwise, where  $\max(Q'_\varepsilon)$  is the vertex with greatest value in  $Q'_\varepsilon$ . The tile  $t'_\varepsilon$  is then defined by:

- $Q'_\varepsilon = Q'_\varepsilon \cup \text{Im}(\varphi_B)$ ,

- $S'_\varepsilon = S'_\varepsilon$ ,
- $\rightarrow'_\varepsilon = \rightarrow'_\varepsilon \cup \{(\varphi_B(n), a, \varphi_B(n')) \mid (n, a, n') \in \rightarrow_B\}$ ,
- $C'_\varepsilon = C'_\varepsilon \cup \{(\varphi_B(n), \lambda) \mid (n, \lambda) \in C_B\}$ ,
- $F'_\varepsilon = F'_\varepsilon \cup \{(t_C, \{( \varphi_B(j), f_C(j)) \mid j \in \text{dom}(f_C)\}) \mid (t_C, f_C) \in F_B\}$ . The update of  $F'$  expresses that the frontier of the new tile  $t'_A$  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. More precisely, we 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 our choice is not to fix this order, enabling the production of several isomorphic semantics for a given RTS.

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

The resulting tile is depicted in Fig. 2 (top). This new tile may be in turn extended by adding a copy of  $t_{\text{comp}}$ , identifying 4, 10 and 11 respectively to 0, 2 and 5. Again, we illustrate the resulting tile in Fig. 2 (bottom) (observe that our definition of  $\varphi_{\text{comp}}$  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.** Let  $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$  be an RTS.  $\mathcal{R}$  defines an IOLTS

$\llbracket \mathcal{R} \rrbracket = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \text{init})$  given by

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

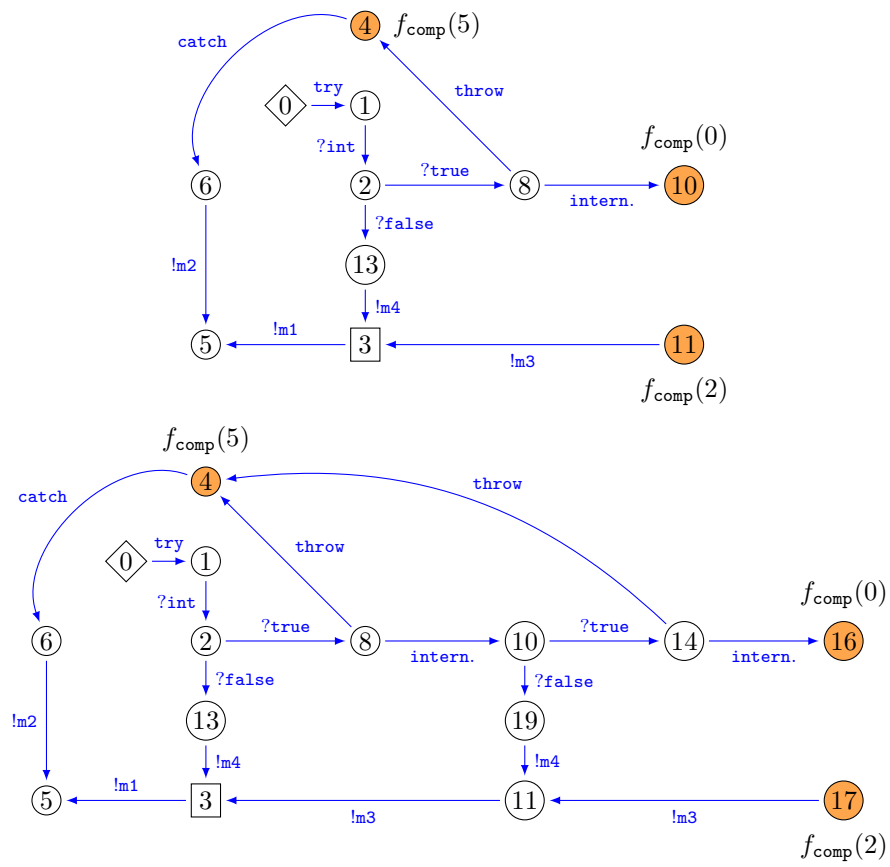
The infinite union of Definition 9 is valid because, by construction, for all  $k \geq 0$ :  $[\mathcal{T}^k(t_0)] \subseteq [\mathcal{T}^{k+1}(t_0)]$ , where  $\subseteq$  is understood as the inclusion of IOLTSs, i.e. inclusion of states, transitions and colourings.

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

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

1. for any state  $q$  of finite degree in  $\llbracket \mathcal{R} \rrbracket$ , 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  $\llbracket v \rrbracket$ ,  $\Gamma_{\llbracket \mathcal{R} \rrbracket}(q) = \Gamma_{\llbracket \mathcal{R} \rrbracket}(q')$ ), thus can be written  $\Gamma_{\llbracket \mathcal{R} \rrbracket}(\llbracket v \rrbracket)$ . Furthermore, we may assume that each vertex possesses a colour reflecting this value (see Corollary 1 below).

**Remark 2.** The IOLTSs obtained from RTSs correspond to the equational, or regular graphs of [10] and [5]. 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 correspond to the frontier, terminal hyperedges are ordinary transitions. Our definition 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.

Figure 2:  $T(t_{\text{main}})$  and  $T^2(t_{\text{main}})$  tiles

### 3.2 Reachability of RTSs

We express, here, fundamental results on RTSs with respect to the formal generation of test suites.

#### Reachability.

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

**Proposition 2** ([5]). *Given an RTS  $\mathcal{R} = ((\Sigma, \Lambda), \mathcal{T}, t_0)$ , a sub-alphabet  $\Sigma' \subseteq \Sigma$ , a colour  $\lambda \in \Lambda$ , and a new colour  $r_\lambda \notin \Lambda$ , an RTS  $\mathcal{R}' = ((\Sigma, \Lambda \cup \{r_\lambda\}), \mathcal{T}', t'_0)$  can be effectively computed, such that  $\llbracket \mathcal{R}' \rrbracket$  is isomorphic to  $\llbracket \mathcal{R} \rrbracket$  with respect to the transitions and the colouring by  $\Lambda$ , and states reachable from a state coloured  $\lambda$  by actions in  $\Sigma'$  are coloured  $r_\lambda$ :  $\mathcal{C}_{\llbracket \mathcal{R}' \rrbracket}(r_\lambda) = \text{reach}_{\llbracket \mathcal{R} \rrbracket}(\mathcal{C}_{\llbracket \mathcal{R} \rrbracket}(\lambda), \Sigma')$ . The same result holds for states co-reachable from  $\lambda$ .*

Proposition 2 is established in [5], performed as a fixed-point computation (propagation of the colours in incremental copies of the tiles). This computation is polynomial. In fact, a naive approach would lead to consider exponentially many tiles. But whenever the original system is formed by a single tile, only polynomially (in the number of elements of the support) different tiles may be reached. Then, from any RTS it is possible to construct a single tile of linear size generating the same system. Thus the actual computation of reachability sets is polynomial in any cases.

Now, Proposition 3.13 (b) of [5] enables to perform several computations (in linear time: in a given tile  $t$  the neighbourhood of each vertex, not in its support, only depends on  $t$ ) related to our purpose. We rephrase it for RTSs.

**Proposition 3** ([5]). *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  $\llbracket \mathcal{R} \rrbracket$  is isomorphic to  $\llbracket \mathcal{R}' \rrbracket$  with respect to the transitions and the colouring by  $\Lambda$ , and every state of  $\llbracket \mathcal{R}' \rrbracket$  of (in- or out- or total-) degree in  $S$  is coloured by  $\#_S$ .*

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

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

### 3.3 $\Sigma_{\mathcal{M}}^\tau$ -closure of RTSs

Abstracting away internal transitions (labelled by actions in  $\Sigma_{\mathcal{M}}^\tau$ ) is important for test generation. While the following proposition shows it is possible to do it for RTSs, we will go slightly further and present precisely how to perform this  $\Sigma_{\mathcal{M}}^\tau$ -closure, and which properties are preserved.

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

This result is classical and follows mainly from [5]: from a given RTS (labelled by  $\Sigma_{\mathcal{M}}^\tau \cup \Sigma^\circ$ ), a context-free grammar generating the same set of traces (in  $\Sigma^{\circ*}$ ) may be constructed, then from such a grammar an RTS of finite degree may be constructed. In order to provide an accurate evaluation of the complexity of the process, and to assess which properties of the original RTS are preserved we provide, now, a direct construction. First we examine carefully which equivalence between IOLTSs may be



preserved. Then we propose a specific way of computing the  $\Sigma_{\mathcal{M}}^{\tau}$ -closure of an IOLTS preserving this equivalence. Since this new computation is non effective for infinite state systems, we eventually explain how to perform it for IOLTSs generated by RTSs, by a direct transformation on RTSs.

### 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 on colours. We thus introduce a notion of coloured traces in order to obtain a finer equivalence, enabling us to establish more precise results.

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

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. Finally, the empty word may not label a coloured trace, this is an arbitrary choice guided by a technical reason: preserving  $\varepsilon$ -labelled coloured traces would be much more difficult and impose a much more complex definition of colours in states.

Let  $\mathcal{M}$  and  $\mathcal{M}'$  be two IOLTSs.  $\mathcal{M}$  and  $\mathcal{M}'$  are *coloured equivalent* whenever they recognise the same coloured traces. This equivalence is more precise than trace equivalence, since two coloured equivalent systems have the same traces (up to the empty word) whereas the converse is not true in general. Conversely coloured equivalence is less precise than bisimulation.

### 3.3.2 Mixed closure.

Our purpose is eventually to compute the closure, with respect to internal events in  $\Sigma_{\mathcal{M}}^{\tau}$ , 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 have difficulties to deal with states generated at the  $V$ -frontier of some tile. Hence we introduce here a general process: *mixed closure* which will be suited to RTSs and furthermore will preserve coloured traces.

Roughly speaking our approach simply consists in adding a new state for each pair of states connected by a  $\Sigma_{\mathcal{M}}^{\tau}$ -labelled transition ( $\Sigma_{\mathcal{M}}^{\tau}$ -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  $\Sigma_{\mathcal{M}}^{\tau}$ , this process will proceed forever. Hence our technique will first eliminate these cycles.

In order to present mixed closure, let us consider IOLTS  $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}}, \text{init}_{\mathcal{M}})$  having  $\Sigma_{\mathcal{M}}^{\tau}$  labelled transitions. Let  $\text{Clo}(\mathcal{M}) = (Q_{\text{Clo}(\mathcal{M})}, \Sigma_{\mathcal{M}}, \Lambda_{\mathcal{M}}, \rightarrow_{\text{Clo}(\mathcal{M})}, \mathcal{C}_{\text{Clo}(\mathcal{M})}, \text{init}_{\mathcal{M}})$  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).

**Strongly connected component of  $\Sigma_{\mathcal{M}}^{\tau}$ -transitions.** Let  $\{C_1, \dots, C_N\}$  be the set of maximal strongly connected component (SCC) of  $\Sigma_{\mathcal{M}}^{\tau}$ -transitions in  $\mathcal{M}$ . Let  $\mathcal{M}_0 \triangleq \mathcal{M}$ , then, for all  $i = 1 \dots N-1$ , 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;

$$\bullet \rightarrow_{i+1} \triangleq \left( \rightarrow_i \setminus \{(q, \mu, q') \mid q, q' \in C_i\} \cup \{(q, \mu, \hat{q}_i) \mid \exists q_\tau \in C_i, \mu \in \Sigma_{\mathcal{M}}, (q, \mu, q_\tau) \in \rightarrow_i\} \right. \\ \left. \cup \{(\hat{q}_i, \mu, q) \mid \exists q_\tau \in C_i, \mu \in \Sigma_{\mathcal{M}}, (q_\tau, \mu, q) \in \rightarrow_i\} \right).$$

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

**Closure of remaining  $\Sigma_{\mathcal{M}}^\tau$ -transitions.** Assume that  $\mathcal{M}$  has been transformed into  $\mathcal{M}_N$  (where  $N$  is the cardinal of SCCs in  $\mathcal{M}$ ), and thus has no strongly connected component of  $\Sigma_{\mathcal{M}}^\tau$ . We now iteratively suppress internal transitions by defining new IOLTS  $\mathcal{M}_{N+1}, \dots, \mathcal{M}_{N+k}$  as follows, starting from  $M_i = M_N$ .

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

- $Q_{i+1} \triangleq Q_i \cup \{q_{\hat{1}2}\}$  with  $q_{\hat{1}2}$  a new state;
- For all states  $q$  of  $Q_{i+1}$ ,  $C_{i+1}(q) = C_i(q_1) \cup C_i(q_2)$  if  $q = q_{\hat{1}2}$ ,  $C_{i+1}(q) = C_i(q)$  otherwise;
- $\rightarrow_{i+1} \triangleq \left( \rightarrow_i \setminus \{(q_1, \tau, q_2)\} \cup \{(q, \mu, q_{\hat{1}2}) \mid q \in Q_i, \mu \in \Sigma_{\mathcal{M}}, (q, \mu, q_1) \in \rightarrow_i\} \right. \\ \left. \cup \{(q_{\hat{1}2}, \mu, q) \mid q \in Q_i, \mu \in \Sigma_{\mathcal{M}}, (q_2, \mu, q) \in \rightarrow_i\} \right).$

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 sequences of internal transitions at each step). In general this may not be applied for infinite states systems.

### 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. Our transformation 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 deal with these problems we will first present a normal form (introduced in [13, 7]), then we explain how to remove 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 once traversing paths (from the support to the  $V$ -frontier) of  $\Sigma_{\mathcal{M}}^\tau$  actions have been removed.

A *path tile* is a tile whose support is only composed of at most a vertices 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  $t_A$ , and any pair of vertices  $v$  and  $v'$  (which may not be distinct) in the support of  $t$ , in constructing a new tile  $t_{Avv'}$  having  $v$  and  $v'$  as support (when they are identical two distinct vertices will be present in  $t_{Avv'}$ ), and respectively  $v$  and  $v'$  with null in and out-degree. The tile  $t_{Avv'}$  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,  $t_{Av0}$  and  $t_{Avi}$ , 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  $\Sigma_{\mathcal{M}}^{\tau}$ -paths between support and  $V$ -frontier.** There are two converse operations which are symmetrical. We only present the one removing internal transitions from the support to the  $V$ -frontier.

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

Let us consider a tile  $t_{Avv'}$  of  $\mathcal{M}$  having  $\Sigma_{\mathcal{M}}^{\tau}$ -paths from  $v$  to elements of the  $V$ -frontier. We iterate the following step to construct a new tile  $t'_{Avv'}$  which will not have any traversing  $\Sigma_{\mathcal{M}}^{\tau}$ -paths, or traversing paths reaching tiles that have already been traversed:

For each  $v''$  in the  $V$ -frontier of  $t_{Avv'}$ , with  $(t_{Bww'}, v_B, v'') \in F_A$ , target of a  $\Sigma_{\mathcal{M}}^{\tau}$ -path from  $v$ , tile  $t_{Avv'}$  with  $t_{Bww'}$ .

Then, for each vertex,  $v_B$ , of the  $V$ -frontier, connected to some tile  $t_{Bww'}$  which is target of a  $\Sigma_{\mathcal{M}}^{\tau}$ -path, identify the vertex,  $v'_B$ , corresponding to the previous occurrence of  $t_{Bww'}$  (which is always possible from the halting condition of previous iteration).

For each transition  $v \xrightarrow{\mu} v_B$ , with  $\ell(v) \geq \ell(v_B)$  (appearing in or after the occurrence of  $t_{Bww'}$ ), add a transition  $v \xrightarrow{\mu} v'_B$ . 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  $\Sigma_{\mathcal{M}}^{\tau}$ -path, the resulting tile, denoted by  $t'_{Avv'}$  has no traversing  $\Sigma_{\mathcal{M}}^{\tau}$ -path. Iterating this process on each tile produces a coloured equivalent RTS with no  $\Sigma_{\mathcal{M}}^{\tau}$ -path.

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

Each of these operations is linear in the total number of vertices (each is visited at most once).

**Removing internal paths.** Now in order to perform the mixed closure we first need to identify the SCCs of internal transitions. Since there are no traversing  $\Sigma_{\mathcal{M}}^{\tau}$ -paths each SCC either fully belongs to a tile or belongs to adjacent tiles. Hence we first perform closure of internal transition between elements of the support: given a tile  $t_A$  having paths of internal transitions between elements of its support, we add extra vertices to the support of  $t_A$  and modifying accordingly tiles having  $t_A$  in their supports. Once it has been performed the converse is done for paths of internal transitions between vertices. Afterwards every mixed closure is performed inside tiles.

First, removal of SCC may be performed in polynomial time. Then, the mixed closure is also polynomial: 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. Since there are no more SCCs, there are at most a quadratic number of sequences of silent actions. Hence a polynomial bound.

**Proposition 5.** *From an RTS  $\mathcal{R}$  with internal actions  $\Sigma^\tau$ , one can effectively compute an RTS  $\text{Clo}(\mathcal{R})$  such that its semantics  $\llbracket \text{Clo}(\mathcal{R}) \rrbracket$  has no internal action, is of finite out-degree, and has the same coloured traces as the IOLTS  $\llbracket \mathcal{R} \rrbracket$ .*

From earlier observations the complete process of performing the closure is of polynomial complexity.

### Weighted RTSs.

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

**Definition 10.** *An RTS  $\mathcal{R}$  with no internal action and with IOLTS semantics  $\llbracket \mathcal{R} \rrbracket = (Q_{\mathcal{R}}, \Sigma, \Lambda, \rightarrow_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}}, \text{init})$  is weighted if  $\mathcal{C}_{\mathcal{R}}(\text{init})$  is a singleton  $\{q_0\}$ , and for any  $w \in \Sigma^*$  and any states  $q, q' \in Q_{\mathcal{R}}$ ,  $q_0 \xrightarrow{w} q$  and  $q_0 \xrightarrow{w} q'$  implies  $\ell(q) = \ell(q')$ : two states reached by the same sequence have the same level.*

Determining whether a given RTS is weighted is decidable (Lemma 4.1 in [7]), in polynomial time. The algorithm initially provided in [13] 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 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  $\llbracket \mathcal{R} \rrbracket$  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 [2], or, more recently, the *weighted grammars* of [6]. 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** ([7]). *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.*

Even though this operation is significantly more technical than for finite state systems, the complexity is also exponential.

**Example 3.** *Assuming internal actions are not observable, we slightly modify the RTS defined in Example 1: 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  $t_{\text{comp}}$ . In the general case some tiles need to be merged first.*

### Synchronous product.

As seen in Section 2.1, the synchronous product of IOLTS 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 (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, we can prove that the product of an RTS with a finite IOLTS is an RTS. More precisely, given any RTS  $\mathcal{R}$  with IOLTS  $\llbracket \mathcal{R} \rrbracket$ , and a finite state IOLTS  $\mathcal{A}$ , one can compute an RTS denoted by  $\mathcal{R} \times \mathcal{A}$  such that  $\llbracket \mathcal{R} \times \mathcal{A} \rrbracket = \llbracket \mathcal{R} \rrbracket \times \mathcal{A}$  (the  $\times$  on the right-hand side of the equality is the product for IOLTSs). This RTS is defined as follows; let  $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma_{\mathcal{A}}, \Lambda_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, \mathcal{C}_{\mathcal{A}}, \text{init}_{\mathcal{A}})$  be a finite IOLTS, and  $\mathcal{R} = ((\Sigma_{\mathcal{R}}, \Lambda_{\mathcal{R}}), \mathcal{T}, t_0)$  be a RTS. The set of tile  $\mathcal{T}_{\mathcal{R} \times \mathcal{A}}$  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  $t_{\mathcal{B}} = ((\Sigma_{\mathcal{B}}, \Lambda_{\mathcal{B}}), Q_{\mathcal{B}}, \rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, S_{\mathcal{B}}, F_{\mathcal{B}})$ , the product tile, denoted by  $t_{\mathcal{B}} \times \mathcal{A}$ , is the following:  $t_{\mathcal{B}} \times \mathcal{A} = ((\Sigma_{\mathcal{B}} \times \Sigma_{\mathcal{A}}, \Lambda_{\mathcal{B}} \times \Lambda_{\mathcal{A}}), Q_{\mathcal{B}} \times Q_{\mathcal{A}}, \rightarrow_{\mathcal{B} \times \mathcal{A}}, \mathcal{C}_{\mathcal{B} \times \mathcal{A}}, S_{\mathcal{B} \times \mathcal{A}}, F_{\mathcal{B} \times \mathcal{A}})$ , with the transitions and colours defined like for products of IOLTS in Section 2.1, the support is simply:  $S_{\mathcal{B}} \times Q_{\mathcal{A}}$ , and, for each  $(t_c, f_c) \in F_{\mathcal{B}}$  ( $f_c : S_c \rightarrow Q_{\mathcal{B}}$ ), there is a  $(t_{c \times \mathcal{A}}, f_{c \times \mathcal{A}})$  with  $t_{c \times \mathcal{A}}$  another tile of the product, and  $f_{c \times \mathcal{A}}$  a fonction between  $S_c \times Q_{\mathcal{A}}$  and  $Q_{\mathcal{B}} \times Q_{\mathcal{A}}$  that associates to any pair  $(q_c, q_{\mathcal{A}})$  the pair:  $(f_c(q_c), q_{\mathcal{A}})$ . 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

Since RTSs are model for recursive systems they may be used to follow an actual execution. Interestingly 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  $v_0 \in \text{init}$ . Let  $T$  be a set of arbitrary symbols denoting tiles and let  $\pi$  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 of the last symbol in  $u_i$ ). We assume 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  $\mu_i$  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  $\mu_i$  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  $\mu_i$  in  $t_i$ ), then assuming that  $t_{i+1}$  is the tile containing a transition labelled  $\mu_i$  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  $\Sigma^*$  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  $\Sigma^*$  (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 implement efficiently runs in systems modelled by RTSs, the system only needs to have access to a single copy of each tile. And a pair formed by the current vertex and a word of  $T^*$ , for each symbolic path 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

In this section and the following, we consider the generation of test cases from RTSs. We focus, here, on weighted RTSs, which are determinizable, and propose an off-line test generation algorithm that operates a selection guided by a test purpose (specified by a finite IOLTS). For a finite IOLTSs  $\mathcal{M}$ , off-line test generation guided by an IOLTS test purpose  $\mathcal{TP}$  consists in a series of operations as follows (see *e.g.* [14]): 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 visible reaction in the specification. Given a specification defined by a RTS  $\mathcal{R}$ , detecting vertices where the 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. We call such paths *divergent*.

**Lemma 1.** *For an RTS  $\mathcal{R}$ , there exists a loop or a divergent path in  $\llbracket \mathcal{R} \rrbracket$  if and only if there exists a vertex  $v$  and two states  $q_1, q_2 \in \llbracket v \rrbracket$  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  $p = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} q_2 \dots$  be an infinite path in  $\llbracket \mathcal{R} \rrbracket$ , with  $\forall k \in \mathbb{N}, \mu_k \in \Sigma^\tau$ . If  $p$  contains a loop, there exists one state of minimal level in this loop, let  $q_{i_1}$  be this state. Now consider an elementary path. As each state is only seen once, we build a sequence of states  $q_{i_k}$  such that  $\forall i_k \leq j, \ell(q_{i_k}) \leq \ell(q_j)$ . As there are only a finite number of vertices, there is a least one  $v$  such that two states of  $\llbracket v \rrbracket$  appear in this path. Let these two states be  $q_1$  and  $q_2$ .

( $\Leftarrow$ ) Suppose that there exist a vertex  $v$  and two states  $q_1, q_2 \in \llbracket v \rrbracket$  such that  $q_1 \xrightarrow{w} q_2$  for  $w \in (\Sigma^\tau)^+$ , and for all states  $q$  on this path,  $\ell(q_1) \leq \ell(q)$ . We distinguish 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  $q_3 \in \llbracket v \rrbracket$ , 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  $\llbracket \mathcal{R} \rrbracket$ : a divergent path.  $\square$

**Proposition 7.** *From any RTS  $\mathcal{R}$ , it is effective to build an RTS denoted  $\Delta(\mathcal{R})$  such that  $\llbracket \Delta(\mathcal{R}) \rrbracket = \Delta(\llbracket \mathcal{R} \rrbracket)$ . Consequently  $\text{Traces}(\llbracket \Delta(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket)$ .*

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

- For output quiescence (deadlock or absence of output), we use Requirement 1, item 2 (defined after Definition 9), which ensures that  $\Gamma_{\llbracket \mathcal{R} \rrbracket}(\llbracket v \rrbracket)$ , for a vertex  $v$  in a tile  $t$  of  $\mathcal{R}$ , has a uniform value. The  $\delta$ -transitions can thus be added to each  $v$  in  $\mathcal{R}$  such that  $\Gamma_{\llbracket \mathcal{R} \rrbracket}(\llbracket v \rrbracket) \subseteq \Sigma_{\mathcal{R}}^?$ . 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. We know that 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  $\mathcal{R}'$ , we proceed as follows:
  - Each vertex  $v$  of tile  $t$  is coloured by a new colour  $\lambda_v$  not in  $\Lambda_{\mathcal{R}'}$ .
  - Proposition 2 is used to colour by  $\lambda'_v$  vertices in  $\text{reach}_{\llbracket \mathcal{R}' \rrbracket}(\Sigma^\tau, \lambda)$ , 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  $\lambda_v$  and  $\lambda'_v$  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  $\text{Traces}(\llbracket \Delta(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket)$ .  $\square$

### 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  $\text{STraces}(\mathcal{R}).\Sigma^{! \delta}$  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  $t_A$ , a new vertex  $v_A^{UnS}$  is added (having colour  $UnS$ ), and new transitions leading to  $v_A^{UnS}$  are added as well for unspecified outputs:

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

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

By construction, we immediately get

$$\text{Traces}_{\mathcal{C}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket) \subseteq \text{STraces}(\llbracket \mathcal{R} \rrbracket). \Sigma_{\mathcal{R}}^{! \delta} \quad (1)$$

$$\text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (2)$$

$$\text{Traces}(\llbracket CS(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket). \Sigma_{\mathcal{R}}^{! \delta} \cup \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (3)$$

The inequality (1) simply says that the traces of sequences recognized in  $UnS$  are suspension traces of  $\mathcal{R}$  prolongedated with outputs. The equality (2) holds because  $\text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket)$ , which are traces of sequences leading outside the colour  $UnS$ , are the original suspension traces of  $\Delta(\mathcal{R})$ , thus  $\text{STraces}(\llbracket \mathcal{R} \rrbracket)$ . 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  $\text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket)$  and  $\text{Traces}_{\mathcal{C}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket)$ , as it can be the projection of both a sequence in  $\Delta(S)$  and a sequence leading to  $UnS$ .

Expanding the definition of  $\text{MinFTraces}(\llbracket \mathcal{R} \rrbracket)$ , we also get

$$\text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) = \text{Traces}_{\mathcal{C}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket) \setminus \text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket CS(\mathcal{R}) \rrbracket) \quad (4)$$

**Remark 3.** Notice that in the introduction of the section, we described a test generation process from finite IOLTS, where a canonical tester was built by output completion to Fail after determinization. A similar process could be described for weighted RTS. However we propose here to first perform an output completion (with a slightly different meaning to a colour  $UnS$ ), and then to determinize (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.

### $\Sigma_{\mathcal{M}}^{\tau}$ -closure

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

$$\text{Traces}_{\mathcal{C}(UnS)}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) \subseteq \text{STraces}(\llbracket \mathcal{R} \rrbracket). \Sigma_{\mathcal{R}}^{! \delta} \quad (5)$$

$$\text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (6)$$

$$\text{Traces}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket). \Sigma_{\mathcal{R}}^{! \delta} \cup \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (7)$$

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

$$\text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) = \text{Traces}_{\mathcal{C}(UnS)}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) \setminus \text{Traces}_{\overline{\mathcal{C}}(UnS)}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) \quad (8)$$

### Canonical tester

Whenever  $Clo(CS(\mathcal{R}))$  is weighted, Proposition 6 enables to determinize it into  $\mathcal{D}(Clo(CS(\mathcal{R})))$ . From  $\mathcal{D}(Clo(CS(\mathcal{R})))$  we build a new RTS  $Can(\mathcal{R})$  called the *canonical tester* of  $\mathcal{R}$  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) we can deduce that

$$\text{Traces}_{\mathcal{C}(\text{Fail})}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) = \text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) \quad (9)$$

$$\text{Traces}_{\overline{\mathcal{C}}(\text{Fail})}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (10)$$

and

$$\text{Traces}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \cup \text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) \quad (11)$$

where the union is now a disjoint union.

In fact

$$\text{Traces}_{\overline{\mathcal{C}}(\text{Fail})}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) = \text{Traces}_{\overline{\mathcal{C}}(\text{Uns})}(\llbracket \text{Clo}(\text{CS}(\mathcal{R})) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \text{ by equality (6)}$$

and

$$\begin{aligned} \text{Traces}_{\mathcal{C}(\text{Fail})}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) &= \text{Traces}_{\mathcal{C}(\text{Uns})}(\llbracket \text{Clo}(\text{CS}(\mathcal{R})) \rrbracket) \setminus \text{Traces}_{\overline{\mathcal{C}}(\text{Uns})}(\llbracket \text{CS}(\mathcal{R}) \rrbracket) \\ &= \text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) \text{ by equality (8)} \end{aligned}$$

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

$$\text{Traces}(\llbracket \text{Can}(\mathcal{R}) \rrbracket) \cap \text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) = \text{Traces}_{\mathcal{C}(\text{Fail})}(\llbracket \text{Can}(\mathcal{R}) \rrbracket)$$

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

**Example 4.** Figure 3, below, represents the canonical tester obtained from Example 1. The vertices labelled by  $F$  correspond to the ones coloured by Fail.

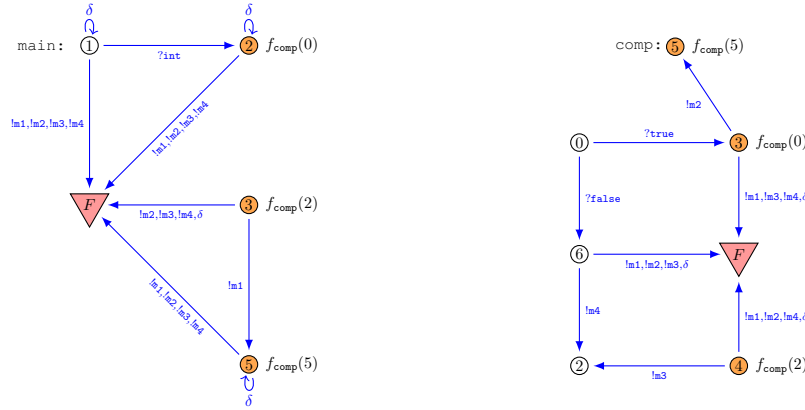


Figure 3: Example of a canonical tester.

### 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 our 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 11.** 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  $\text{Can}(\mathcal{R})$  and  $\mathcal{TP}$  is an RTS. On this product, new colours are specified as follows :

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

Note that, by construction, each vertex has a unique colour in  $\{\text{Fail}, \text{Pass}, \text{None}, \text{Inc}\}$ . 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. 3) resulting from Example 1, Figure 4 depicts the test case obtained with the test purpose,  $\mathcal{TP}$ , accepting the traces in  $(\Sigma^{\circ\delta})^* \cdot ?\text{true} \cdot ?\text{true} \cdot (\Sigma^{\circ\delta})^* \cdot !\text{m1}$ . 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  $?\text{true}$  and  $!\text{m1}$ , 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 4. For a better readability, the only vertices represented are those accessible from  $(1, q_1)$  (of tile *main*) furthermore, in each tile, vertices coloured by *Fail* (resp., *Inc*) are merged.

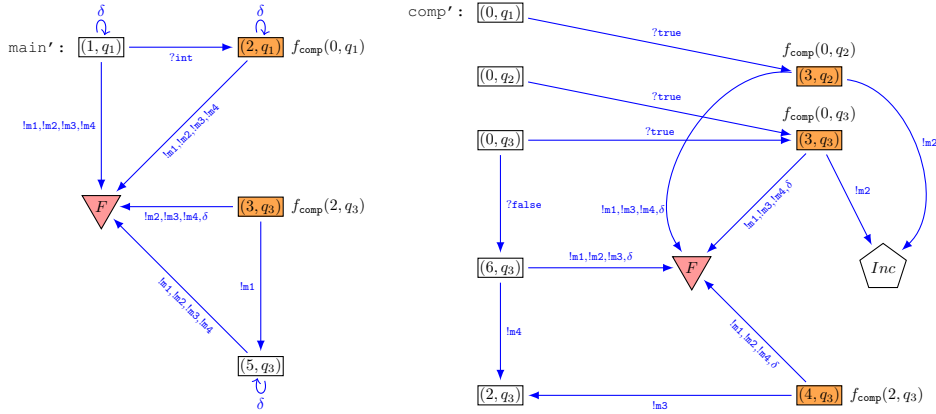


Figure 4: Example of a test case

## 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.

The most computationally heavy operation, like for finite state systems, is an exponential determinization. Before this operation two polynomial ones are performed to suspend, and complete the system. Afterwards, checking whether the computed RTS is weighted is also polynomial. This is summarized in the following list.

- 1) Computation of  $\Delta(\mathcal{R})$ : **polynomial**
- 2) Computation of  $CS(\mathcal{R})$ : **polynomial**
- 3) Computation of  $Clo(CS(\mathcal{R}))$ : **polynomial**
- 4) Checking whether  $Clo(CS(\mathcal{R}))$  is weighted: **polynomial**
- 5) Determinizing  $Clo(CS(\mathcal{R}))$ : **exponential**
- 6) Synchronous product  $Can(\mathcal{R}) \times \mathcal{TP}$  and selection: **polynomial**

Observe that the computational complexities of these operations are similar to those for finite state systems. Furthermore the blow-up arising from determinization is in some sense unavoidable: testing, exhaustively, a non-deterministic, system of size  $k$ , with no loop, might involve  $2^k$  distinct tests. Hence the size of the canonical tester is polynomial with respect to the expected number of generated tests.

### 4.3 Properties of generated test cases

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

#### Soundness and strictness

According to the construction of  $\mathcal{P} = Can(\mathcal{R}) \times \mathcal{TP}$ , the definition of  $\mathcal{C}_p(Fail)$ , and pruning, selection by  $\mathcal{TP}$  do not add any colouring by *Fail* with respect to  $Can(\mathcal{R})$ , thus  $Traces_{\mathcal{C}(Fail)}(\llbracket \mathcal{TC} \rrbracket) = Traces(\llbracket \mathcal{TC} \rrbracket) \cap Traces_{\mathcal{C}(Fail)}(\llbracket Can(\mathcal{R}) \rrbracket)$ . By equation (9) we deduce  $Traces_{\mathcal{C}(Fail)}(\llbracket \mathcal{TC} \rrbracket) = Traces(\llbracket \mathcal{TC} \rrbracket) \cap MinFTraces(\llbracket \mathcal{R} \rrbracket) \subseteq MinFTraces(\llbracket \mathcal{R} \rrbracket)$  which proves both strictness (equality) and soundness (inclusion).

#### Exhaustiveness

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

Let  $\sigma' = \sigma.a \in MinFTraces(\llbracket \mathcal{R} \rrbracket) = Traces_{\mathcal{C}(Fail)}(\llbracket Can(\mathcal{R}) \rrbracket)$  be a minimal non-conformant trace for  $\mathcal{R}$ . We have  $\sigma \in STraces(\llbracket \mathcal{R} \rrbracket)$  and there exists  $b \in \Sigma^{!\delta}$  such that  $\sigma.b \in STraces(\llbracket \mathcal{R} \rrbracket)$  (if no output continues  $\sigma$  in  $STraces(\llbracket \mathcal{R} \rrbracket)$ , a  $\delta$  does). Now, it remains to define a test purpose  $\mathcal{TP}$  such that  $\sigma.b \in Traces_{\mathcal{C}(Accept)}(\mathcal{TP})$ . Let  $\mathcal{TC}$  be the test case generated from  $\mathcal{R}$  and  $\mathcal{TP}$ . By construction of  $\mathcal{TC}$  from  $\mathcal{R}$  and  $\mathcal{TP}$ , we will get both  $\sigma.b \in Traces(\llbracket \mathcal{TC} \rrbracket)$  and  $\sigma' \in Traces_{\mathcal{C}(Fail)}(\llbracket \mathcal{TC} \rrbracket)$ . 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. Formally,

**Definition 12.** A test case  $\mathcal{TC}$  is precise with respect to an IOLTS specification  $\mathcal{M}$  and a test purpose  $\mathcal{TP}$  if  $\text{Traces}_{\mathcal{C}(Pass)}(\llbracket \mathcal{TC} \rrbracket) = \text{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \text{STraces}(\mathcal{M}) \cap \text{Traces}(\llbracket \mathcal{TC} \rrbracket)$ .

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  $\text{Can}(\mathcal{R})$ . Thus  $\text{Traces}_{\mathcal{C}(Pass)}(\llbracket \mathcal{TC} \rrbracket) = \text{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \text{STraces}(\llbracket \mathcal{R} \rrbracket)$ , which (since  $\text{Traces}_{\mathcal{C}(Pass)}(\llbracket \mathcal{TC} \rrbracket) \subseteq \text{Traces}(\llbracket \mathcal{TC} \rrbracket)$ ) 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 [20] 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. In this section we will present this technique, and establish 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  $\sigma^\tau$ -closure. The process thus starts from the closure of the output-completed specification  $\text{Clo}(CS(\mathcal{R}))$  defined in Section 4. This time, the canonical tester cannot be built in general by determinization from  $\text{Clo}(CS(\mathcal{R}))$ . However, using Proposition 4, one can build  $\text{Clo}(CS(\mathcal{R}))$  from  $\mathcal{R}$ , ensuring the following properties of equations (5), (6), (7) and (8):

$$\text{Traces}_{\mathcal{C}(UnS)}(\llbracket \text{Clo}(CS(\mathcal{R})) \rrbracket) \subseteq \text{STraces}(\llbracket \mathcal{R} \rrbracket, \Sigma_{\mathcal{R}}^{\delta}) \quad (5)$$

$$\text{Traces}_{\bar{\mathcal{C}}(UnS)}(\llbracket \text{Clo}(CS(\mathcal{R})) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (6)$$

$$\text{Traces}(\llbracket \text{Clo}(CS(\mathcal{R})) \rrbracket) = \text{STraces}(\llbracket \mathcal{R} \rrbracket, \Sigma_{\mathcal{R}}^{\delta}) \cup \text{STraces}(\llbracket \mathcal{R} \rrbracket) \quad (7)$$

and

$$\text{MinFTraces}(\llbracket \mathcal{R} \rrbracket) = \text{Traces}_{\mathcal{C}(UnS)}(\llbracket \text{Clo}(CS(\mathcal{R})) \rrbracket) \setminus \text{Traces}_{\bar{\mathcal{C}}(UnS)}(\llbracket \text{Clo}(CS(\mathcal{R})) \rrbracket) \quad (8)$$

#### 5.1.1 Product and colouring

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

- $\mathcal{C}_{\mathcal{P}}(UnS) = \mathcal{C}_{\text{Clo}(CS(\mathcal{R}))}(UnS) \times \mathcal{Q}_{\mathcal{TP}}$
- $\mathcal{C}_{\mathcal{P}}(Pass) = \bar{\mathcal{C}}_{\text{Clo}(CS(\mathcal{R}))}(UnS) \times \mathcal{C}_{\mathcal{TP}}(Accept)$
- $\mathcal{C}_{\mathcal{P}}(None) = \text{Coreach}(\mathcal{C}_{\mathcal{P}}(Pass)) \setminus \mathcal{C}_{\mathcal{P}}(Pass)$
- $\mathcal{C}_{\mathcal{P}}(Inc) = \mathcal{Q}_{\mathcal{P}} \setminus (\mathcal{C}_{\mathcal{P}}(Fail) \cup \mathcal{C}_{\mathcal{P}}(Pass) \cup \mathcal{C}_{\mathcal{P}}(None))$

### 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 way similar to [20]. 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  $(\Sigma^{o\delta})^* \cdot (\{Fail, Pass, None, Inc\} \cup \{\varepsilon\})$ . Given a tree  $\theta$ , for some symbol  $a$ , we introduce the notation  $a; \theta \triangleq \{au \mid u \in \theta\}$ . Furthermore, given two trees  $\theta, \theta'$ , 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  $\llbracket Clo(CS(\mathcal{R})) \rrbracket$ . Let us define test cases by applying the following algorithm recursively, starting from the initial state  $\mathcal{C}_{\mathcal{P}}(init)$ . Choose non deterministically between one of the following operations.

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

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

with:

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

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

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 from 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  $Traces(\llbracket Clo(CS(\mathcal{R})) \rrbracket) \setminus Traces_{\overline{\mathcal{C}}(UnS)}(\llbracket Clo(CS(\mathcal{R})) \rrbracket) = MinFTraces(\llbracket \mathcal{R} \rrbracket)$ . Thus  $Traces_{\mathcal{C}(Fail)}(\mathcal{TC}) = MinFTraces(\llbracket \mathcal{R} \rrbracket) \cap Traces(\mathcal{TC})$  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  $\text{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \text{Traces}_{\mathcal{C}(Pass)}(\text{Clo}(CS(\mathcal{R})) \times \mathcal{TP}) \cap \text{Traces}(\mathcal{TC})$ . Then, by definitions of the colours, we obtain:  $\text{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \text{Traces}_{\overline{\mathcal{C}}_{UMS}}(\text{Clo}(CS(\mathcal{R}))) \cap \text{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \text{Traces}(\mathcal{TC})$ . Which eventually proves precision:  $\text{Traces}_{\mathcal{C}(Pass)}(\mathcal{TC}) = \text{STraces}(\mathcal{R}) \cap \text{Traces}_{\mathcal{C}(Accept)}(\mathcal{TP}) \cap \text{Traces}(\mathcal{TC})$ .

## 5.3 Application of on-line test cases generation

In previous subsections we have seen 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  $T^*$ ).

**Example 6.** *Applying this construction for the RTS defined in Example 1 (which is deterministic but will be suited for the purpose of illustration). Let us define  $T \triangleq \{m, c\}$  where the symbols represents respectively tiles `main` and `comp`. Assume  $8, true, true, true$  is the sequence of input already computed. We 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*, we reach  $(6, mcccc)$  then expect the output "You stopped" (message `m4`), 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 the "Done" (message `m1`) from the implementation. This would be the end of this test.*

## 6 Conclusion

In this paper we have presented recursive tile systems, a general model of IOLTS allowing for recursion. We have provided algorithms to produce sound, strict and exhaustive test suites, either off-line or on-line. These algorithms enable to employ test purposes (even, for the on-line case) which are a classical way to drive tests towards sensitive properties. We have also established the precision of our tests with respect to test purposes. Moreover we have provided precise assessments of the complexities of involved operations. In fact even though our approach enables to model infinite state objects, the algorithms are not significantly more costly.

Our 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 RTS [11, 3]. This would enable to take into account quantitative properties of systems, or to express coverage properties of finite test suites.

## References

- [1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *13th International Conference on Computer Aided Verification, (CAV'01)*, volume 2102 of *LNCS*, pages 207–220, 2001.
- [2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)*, pages 202–211. ACM, 2004.
- [3] N. Bertrand and C. Morvan. Probabilistic regular graphs. In *12th International Workshop on Verification of Infinite-State Systems, INFINITY'10*, volume 39 of *EPTCS*, pages 77–90, Singapour, August 2010.
- [4] Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed automata. *Logical Methods in Computer Science*, 8(4), 2012.
- [5] D. Caucal. Deterministic graph grammars. In *Texts in logics and games 2*, pages 169–250, 2007.
- [6] D. Caucal. Synchronization of regular automata. In *34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, volume 5734 of *LNCS*, pages 2–23, 2009.
- [7] D. Caucal and S. Hassen. Synchronization of grammars. In *Third International Computer Science Symposium in Russia (CSR'08)*, volume 5010 of *LNCS*, pages 110–121, 2008.
- [8] S. Chédor, C. Morvan, S. Pinchinat, and H. Marchand. Analysis of partially observed recursive tile systems. In *11th edition of Workshop on Discrete Event Systems*, pages 265–271, Guadalajara, Mexico, September 2012.
- [9] C. Constant, B. Jeannet, and T. Jéron. Automatic test generation from interprocedural specifications. In *TestCom/FATES'07*, volume 4581 of *LNCS*, pages 41–57, 2007.
- [10] B. Courcelle. *Handbook of Theoretical Computer Science*, chapter Graph rewriting: an algebraic and logic approach. Elsevier, 1990.
- [11] J. Esparza, A. Kučera, and R. Mayr. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science*, 2(1), 2006.
- [12] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A symbolic framework for model-based testing. In *FATES 2006 and RV 2006, Revised Selected Papers*, volume 4262 of *LNCS*, pages 40–54, 2006.
- [13] S. Hassen. *Synchronisation de grammaires de graphes*. Phd thesis, Université de la Réunion, 2008.
- [14] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.
- [15] B. Jeannet, T. Jéron, and V. Rusu. Model-based test selection for infinite-state reactive systems. In *5th International Symposium on Formal Methods for Components and Objects (FMCO'06), Revised Lectures*, volume 4709 of *LNCS*, pages 47–69, 2006.
- [16] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

- 
- [17] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using Uppaal. In *Formal Approaches to Software Testing (FATES'04)*, volume 3395 of *LNCS*, pages 79–94, 2005.
  - [18] D. Nowotka and J. Srba. Height-deterministic pushdown automata. In *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*, volume 4708 of *LNCS*, pages 125–134, 2007.
  - [19] G.J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.
  - [20] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.





**RESEARCH CENTRE  
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu  
35042 Rennes Cedex

Publisher  
Inria  
Domaine de Voluceau - Rocquencourt  
BP 105 - 78153 Le Chesnay Cedex  
[inria.fr](http://inria.fr)

ISSN 0249-6399