Tree Pattern Rewriting Systems[⋆]

B. Genest[3], A. Muscholl[1], O. Serre[2], and M. Zeitoun[1]

[1]LaBRI, Bordeaux; [2]LIAFA, Paris 7 & CNRS; [3]IRISA, Rennes 1 & CNRS

Abstract. Classical verification often uses abstraction when dealing with data. On the other hand, dynamic XML-based applications have become pervasive, for instance with the ever growing importance of web services. We define here *Tree Pattern Rewriting Systems* (TPRS) as an abstract model of dynamic XML-based documents. TPRS systems generate infinite transition systems, where states are unranked and unordered trees (hence possibly modeling XML documents). Their guarded transition rules are described by means of tree patterns. Our main result is that given a TPRS system $(T, \mathcal{R})$, a tree pattern P and some integer k such that any reachable document from T has depth at most k, it is *decidable* (albeit of non elementary complexity) whether some tree matching P is reachable from T.

1 Introduction

Classical verification techniques often use abstraction when dealing with data. On the other hand, dynamic *data-intensive* applications have become pervasive, for instance with the ever growing importance of web services. The format of the data exchanged by web services is based on XML, which is nowadays the standard for semistructured data. XML documents can be seen as unranked trees, *i.e.* trees in which every node can have an arbitrary (but finite) number of children, not depending on its labels. Very often, the order of siblings in the document is of no importance. In this case, trees are in addition unordered. There is a rich body of results concerning the analysis of fixed XML documents (with or without data), see e.g [13,11] for surveys on this topic.

The analysis of the *dynamics* of XML documents accessed and updated in a multi-peer environment has been considered only very recently [2,3]. Dynamically evolving XML documents are of course crucial, for instance when doing static analysis of XML-based web services. A general framework, Active XML (*AXML* for short), has been defined in [2] to unify data (XML) and control (services), by allowing data to be given implicitly in form of service calls.

In this paper we propose an abstract model for dynamically evolving documents, based on guarded rewriting rules on unranked, unordered tree. We show that basic properties, such as reachability of tree patterns and termination, are decidable for a natural subclass of our rewriting systems.

A standard technique to analyze unranked trees is to encode them as binary trees [13]. However, this encoding does not preserve the depth of the tree,

[⋆] Work supported by ANR DocFlow, ANR DOTS and CREATE ACTIVEDOC.

neither locality, nor path properties. For these reasons, we define guarded tree rewriting systems directly on unranked trees. The rewriting rules are based on tree patterns, that occur in two distinct contexts. First, tree patterns are used for describing how the structure of the tree changes through the rules: subtrees can be moved or deleted, and new nodes can be added. Thus, documents evolve in a *non monotonous* way. Second, rules are guarded, and the guard condition is tested via a *Tree Pattern Query* (TPQ). The role of the TPQ is actually twofold: it is used in the pre-condition of the rule, and the query results can enhance the information of the new tree. We call such systems *Tree Pattern Rewriting System, TPRS* for short. For an easier comparison with other works, we include an example of a Mail-Order System in our presentation, close to the one used in [3].

The main tool we use to show decidability of various properties of TPRS are well-structured systems [1,8]. Such systems cover several interesting classes of infinite-state systems, such as Petri nets and lossy channel systems. Our TPRS are of course not well-structured, in general. We impose two restrictions in order to obtain well-structure. First, guards must be used positively: equivalently, a rule cannot be disabled because of the existence of some tree pattern. Second, we need a uniform bound on the depth of the trees obtained by rewriting. Indeed, we show that if the depth is not uniformly bounded, then TPRS can encode Turing machines. Notice that the depth restriction is very realistic in the XML setting, since such documents are usually large, but shallow. We show that TPRS that satisfy both conditions yield well-structured transition systems, and we show how to apply forward and backward analysis of well-structured systems for obtaining the decidability of pattern reachability as well as termination. On the negative side, we show that exact reachability, confluence and the finite state property are undecidable for such TPRS. One can notice that the reachability of a given tree pattern is more likely to be useful in practice than exact reachability, that supposes the complete knowledge about the target document. In the decidable cases, we also show that the complexity is at least non elementary.

Related work. We review here other approaches where it is possible to decide behavioral properties of active documents.

The systems called *positive AXML* in [2] are *monotonous*: a document is modified by adding subtrees at nodes labeled by service calls, deletions are not possible. In particular, trees can only grow, which is not the case for the TPRS defined here. For instance, for the mail order example this means that product cannot be deleted from the cart. Moreover, there is no deterministic description of the semantics of a service: a service call can create any tree granted that it satisfies some DTD. Such a system is always confluent, and one can decide whether, after some finite number of steps, the system will stabilize [2].

Guard AXML [3] is very similar to our model, service calls being based on tree pattern queries. The focus of [3] is to analyze the action of non recursive services over documents satisfying a given schema, and with potentially unbounded data (trees are labeled by symbols from an infinite alphabet and tree patterns use data constraints). Compared with our framework, [3] uses more powerful guards, namely boolean combinations of tree patterns guards. However, the price to pay is that decidability results in [3] require a uniform bound on the length of the rewriting chains. In contrast, our TPRS model active documents with possibly recursive service calls.

A seemingly related area are term rewriting systems modulo associativity and commutativity [10]. However, these rewriting systems act on *ranked trees*, so applying results from this area on unranked trees requires to work on some ranked encoding of the tree. Also, it is not clear how to simulate e.g. TPRS rules that move all but some specific subtrees of a given node by term rewriting rules. Tree rewriting on unranked (ordered) trees has been considered in [12]. The difference to our setting is that the rewriting is *ground*, i.e., rules can only be applied at the deepest levels of the tree, which makes reachability decidable.

2 Tree Pattern Rewriting

The tree rewriting model presented in this section is inspired by the Active XML (AXML) system developped at INRIA [4]. Active XML extends the framework of XML to describe semi-structured data by a dynamic component, describing data implicitly via service (function) calls. The evaluation of such calls is based on queries, resulting in extra data that can be added to the document tree. The abstract model is that of XML, *i.e.*, unranked, unordered, labeled trees, together with a specification of the semantics for each service.

Trees considered in this paper are labeled by tags from a finite set $\mathcal{T}$. We will distinguish a subset $\mathcal{T}_{var} \subseteq \mathcal{T}$ of so-called *tag variables.* In addition, we use the special symbol \$ to mark nodes where service calls insert new data. Trees are in the following unranked and unordered, with nodes labeled by $\mathcal{T} \cup \mathcal{T}_{\$}$, where $\mathcal{T}_{\$} = \mathcal{T} \times \{\$\}$. We will not distinguish function/service nodes, since we consider here an abstract model for AXML documents, that is based on tree rewriting. We also do not consider multiple peers actually: their joint behavior can be described as the evolution of a unique document tree.

A *tree* $(V, \mathsf{parent}, root, \lambda)$ consists of a set of nodes V with a distinguished node called *root*, a mapping $\mathsf{parent} : V \setminus \{root\} \to V$ associating a node with its parent, and a mapping $\lambda : V \to \mathcal{T} \cup \mathcal{T}_{\$}$ labeling each node by a tag. Moreover,

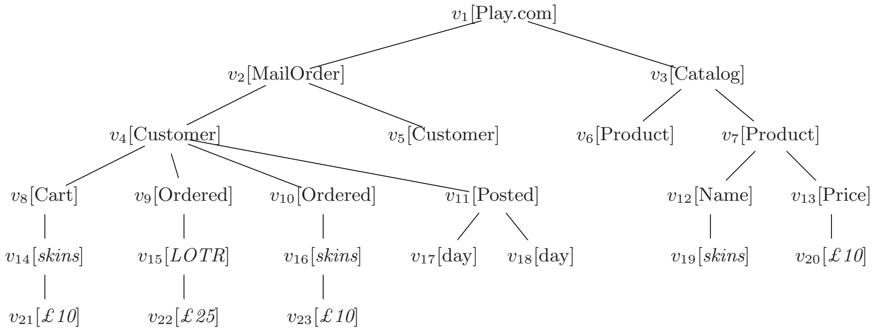

Fig. 1. Tree document representing a catalog of products and customers history.

for each node $v \in V$, there is some $k \geq 0$ such that $\mathsf{parent}^k(v) = \mathit{root}$. Such a tree is called a *document* if its labeling satisfies $\lambda(V) \subseteq \mathcal{T} \setminus \mathcal{T}_{var}$, that is, no node label uses a tag variable or the \$ sign. A *forest* is a finite multiset of trees.

Consider as an example the tree in Fig. 1. Informally, this tree represents a simplified version of the `Play.com` database, containing several products and information about customers. The subtrees of nodes v_5 and v_6 are not represented in the figure. The document shows two customers, one of which is currently shopping on the website with one product in her cart. This customer has 3 outstanding orders, one of which was posted 2 days ago (the counter *days* is encoded in unary in the tree). One can represents a tree in a term-like way. For example, to denote the empty catalog with no customer, we write $v_1[\text{Play.com}](v_2[\text{MailOrder}], v_3[\text{Catalog}])$, or if node names are irrelevant, $[\text{Play.com}]([\text{MailOrder}], [\text{Catalog}])$. Since trees are unordered, a tree can have several such representations.

The atomic operations in our model are from a set $\mathcal{R}$ of guarded tree rewriting rules, as described below. On an abstract level we view a service s as described for instance by a regular expression $R(s)$ over the set of rewriting rules $\mathcal{R}$. For example, the following expression describes an order service on `Play.com`: $(\mathsf{add\text{-}product} + \mathsf{delete\text{-}product}\ )^*\mathsf{checkout}$.

The tree resulting in the invocation of service s corresponds to the application of some sequence of rewriting rules in $R(s)$. The atomic rewriting rules will use queries based on tree patterns (the descendant relation can be used together with the child relation), as described next. The symbol $\uplus$ used below stands for the disjoint union.

Definition 1 (Tree-Pattern). *A tree pattern (TP for short) is a tuple* $P = (V, \mathsf{parent}, \mathsf{ancestor}, \mathit{root}, \lambda)$*, where* $(V, \mathsf{parent} \uplus \mathsf{ancestor}, \mathit{root}, \lambda)$ *is a tree.*

A tree pattern represents a set of trees that have a similar shape. As for trees, a TP can be described in a term-like way, $\mathsf{ancestor}$-edges being represented by the symbol – (such edges are represented by a double line in the figures). For instance, the tree pattern LQBill shown in Fig. 2 can be written as $w_1(-w_2(w_3(w_4)))$, with $\lambda(w_1) = \text{Play.com}$, $\lambda(w_2) = \text{Ordered}$, $\lambda(w_3) = X$, $\lambda(w_4) = Y$ (here X and Y are tag variables: $X, Y \in \mathcal{T}_{var}$). This pattern represents trees with root "Play.com", having a node "Ordered", having itself a grandchild.

Definition 2 (Matching). *A tree* $T = (V, \mathsf{parent}, \lambda, \mathit{root})$ *matches a TP* $P = (V', \mathsf{parent}', \mathsf{ancestor}', \lambda', \mathit{root}')$ *if there exist two mappings* $f : V' \rightarrow V$ *and* $t : \mathcal{T}_{var} \rightarrow \mathcal{T} \setminus \mathcal{T}_{var}$ *such that:*

- $f(\mathit{root}') = \mathit{root}$,
- *For all* $v \in V'$, $\lambda(f(v)) = \lambda'(v)$ *if* $\lambda'(v) \notin \mathcal{T}_{var}$*, and else* $\lambda(f(v)) = t(\lambda'(v))$,
- *For all* $v \in V'$ *the following holds*
  - *If* $\mathsf{parent}'(v)$ *is defined, then* $f(\mathsf{parent}'(v)) = \mathsf{parent}(f(v))$*;*
  - *If* $\mathsf{ancestor}'(v)$ *is defined, then* $f(\mathsf{ancestor}'(v))$ *is an ancestor of* $f(v)$ *in* T.

Mappings (f, t) *as above are called a* matching *between* T *and* P*. Furthermore, if* f *is injective, then* (f, t) *is called an* injective matching.

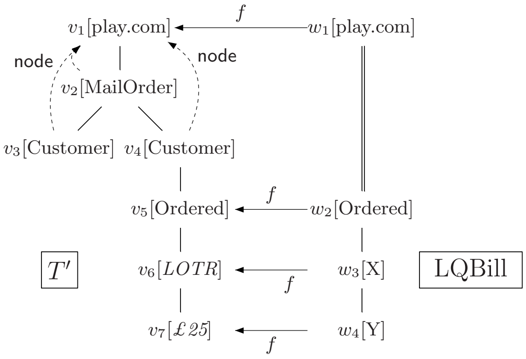

Fig. 2. A tree T' matching the TP LQBill.

Fig. 2 shows an example of an injective matching between a tree T' and the TP LQBill. The only possible matching is $f(w_1) = v_1, f(w_2) = v_5, f(w_3) = v_6, f(w_4) = v_7$ and $t(X)$ =*LOTR*, $t(Y)$ = *£25*. With a matching $f : V' \to V$ we associate the mapping $\mathsf{node} : V \setminus f(V') \to f(V')$ with $\mathsf{node}(v)$ being the lowest ancestor of v in $f(V')$. For instance, for the mapping f matching T to P, $\mathsf{node}(v_2) = \mathsf{node}(v_3) = \mathsf{node}(v_4) = v_1$.

Similarly to [2] we use in our model tree-pattern queries (TPQ for short, also called *positive queries* in [2]). Such queries have the form $Q \rightsquigarrow P$, with Q a TP and P a tree, and the variables used in Q are also used in P. The TP Q selects tags in the tree. The result of a $\mathsf{query} = Q \rightsquigarrow P$ on T is the forest $\mathsf{query}(T)$ of all instantiations of P by matchings between T and Q. That is, for each matching (f, t) between T and Q we obtain an instance of P in which each $\mathcal{T}_{var}$-label X has been replaced by the tag $t(X)$. For instance, let RQBill be the tree Product (Name(X),Price(Y)). Then the result of the TPQ (LQBill $\rightsquigarrow$ RQBill) on the tree in Fig. 1 is the forest depicted in Fig. 3.

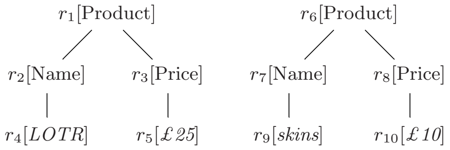

Fig. 3. Result forest of the TPQ *LQBill* $\rightsquigarrow$ *RQBill* on document T.

We now define a generic kind of (guarded) rewriting rules, as a model for active documents. Our rules are based on tree patterns, that occur in two distinct contexts. First, tree patterns are used for describing how the structure of the document tree changes through the rule - some subtrees might be deleted, new nodes can be added. Second, rules are guarded, and the guard condition is tested via a TPQ. The role of the query is actually twofold: it is used in the pre-condition of the rule, and its result can enhance the information of the new tree.

Definition 3 (TP rules). *A TP rule is a tuple* $(\mathsf{left}, \mathsf{query}, \mathsf{guard}, \mathsf{right})$*, such that:*

- left *is a TP* $(V_l, \mathsf{parent}_l, \mathsf{ancestor}_l, \lambda_l, root_l)$ *over* $\mathcal{T}$*,*
- right *is a TP* $(V_r, \mathsf{parent}_r, \mathsf{ancestor}_r, \lambda_r, root_r)$ *over* $\mathcal{T} \cup \mathcal{T}_\$$*,*
- query *is a TPQ,*
- guard *is a set of forests.*

We require the following additional properties:

1. *all tag variables used in* right *appear also in* left*, and*
2. $\mathsf{ancestor}_r(x) = y$ *iff* $x, y \in V_l \cap V_r$ *and* $\mathsf{ancestor}_l(x) = y$*.*

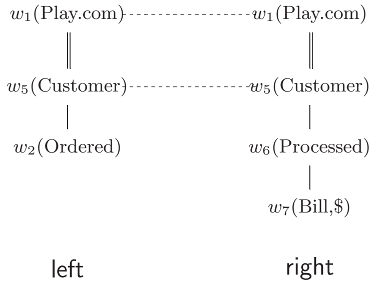

Fig. 4. Tree patterns left and right of a rule.

The additional conditions on right ensure that the right-hand side of a TP rule determines the form of the resulting tree, as it is explained below. For instance, Bill = $(\mathsf{left}, (\text{LQBill} \rightsquigarrow \text{RQBill}), \mathsf{guard}, \mathsf{right})$ is a TP rule, with $\mathsf{left}, \mathsf{right}$ defined as in Fig. 4. Informally, the rule says that the system will process a Bill of the current order, and will tag the order as processed. The guard guard will be usually specified as a finite set of trees. In this case, the guard is fulfilled if the result of the query covers one of the tree of guard (see the decidable section 4).

We first describe the semantics of a rule using the rule Bill as an example on the tree in Fig. 1. First, we compute an *injective* mapping f which maps the nodes w_1, w_5, w_2 of left with the nodes v_1, v_4, v_{10} of T, respectively. We produce a new tree by rearranging and relabeling the nodes of T in the image of f, that is v_1, v_4, v_{10}. Some nodes can be deleted and others created. The resulting tree is shown in Fig. 5. We keep all nodes of T which are matched by nodes of left also present in right (v_1 and v_4), as well as their descendants by node^{-1}. That is, we keep all nodes labeled by v_i in Fig. 5. In particular, a node matched in left which does not appear in right is deleted (as v_{10}, matched to w_2), as well as its node^{-1} descendants (v_{16} and v_{23}). The TP right makes it possible to create new nodes, present in right but not in left, as w_6, w_7. In addition, the TPQ query of the rule is used to attach a copy of the returned forest to all \$-marked nodes of right. Furthermore, if the TPQ $Q \rightsquigarrow P$ uses in Q node name m common to

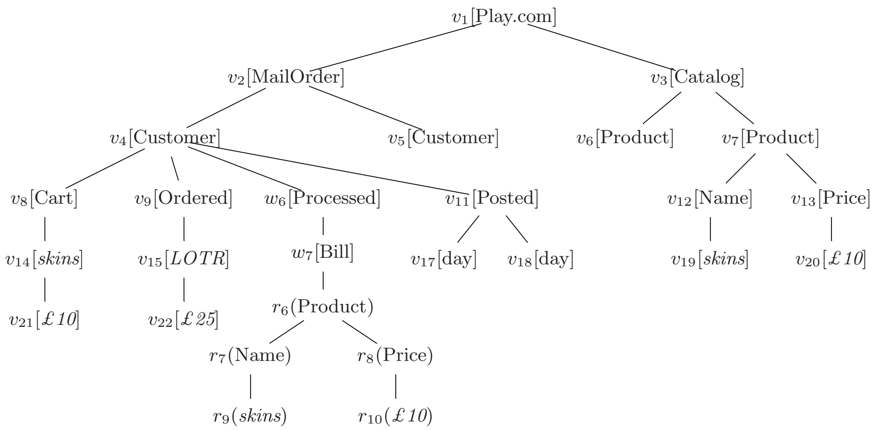

Fig. 5. The tree document T' resulting of the application of the rule Bill.

left, then the results of the TPQ are restricted to those where m matches $f(m)$. For instance, in the TP rule Bill, the TP LQBill uses names w_1, w_2 common to left, hence the results of the TPQ (LQBill $\rightsquigarrow$ RQBill) are restricted to the particular order chosen by the matching f between T and left. The result is thus the subtree rooted at node r_6 in Fig. 3, but not the subtree rooted at r_1, since it would require $f(w_2) = v_9$, while $f(w_2) = v_{10}$. This restriction is desirable, since we want to process a bill only for the products of this particular order. Here, w_7 is \$-marked, and the result forest is defined by nodes $r_6, \cdots r_{10}$.

More formally, let $\mathsf{query} = Q \rightsquigarrow P$ be a TPQ and let (f,t) be an injective matching between T and left. Moreover, let V_l be the nodes of left and V_Q those of Q. Let $S_1, \ldots, S_k$ be the trees composing the resulting forest $\mathsf{query}(T)$, and let $g_1, \ldots, g_k$ be the respective associated matchings (that is, $g_i : V_Q \rightarrow V$ and S_i is the instantiation of P by g_i). Then we define $\mathsf{query}_f(T)$ as the forest $S_{i_1}, \ldots, S_{i_l}$ of those trees S_j such that g_j agrees with f over $V_l \cap V_Q$. That is, $\mathsf{query}_f(T)$ is the subset of $\mathsf{query}(T)$ that is consistent with the matching f. We now turn to the formal semantics of rules.

Definition 4 (Semantics of rules). *Let $T = (V, \mathsf{parent}, \lambda, root)$ be a tree and $R = (\mathsf{left}, \mathsf{query} = Q \rightsquigarrow P, \mathsf{guard}, \mathsf{right})$ be a rule, where $\mathsf{left} = (V_l, \mathsf{parent}_l, \mathsf{ancestor}_l, \lambda_l, root_l)$ and $\mathsf{right} = (V_r, \mathsf{parent}_r, \mathsf{ancestor}_r, \lambda_r, root_r)$.*

We say that R is enabled *if there exists an injective matching (f,t) from left into T, such that $\mathsf{query}_f(T) \in \mathsf{guard}$. The result of the application of R via (f,t) is the tree $T' = (V', \mathsf{parent}', \lambda', root')$ defined as follows:*

- *$V' = V_1 \uplus V_2 \uplus V_3 \uplus V_4$ with*
  1. *$V_1 = f(V_l \cap V_r)$, % in the example on Fig. 5, $V_1 = \{v_1, v_4\}$*
  2. *$V_2 = \mathsf{node}^{-1}(V_1)$, % in the example on Fig. 5, $V_2 = \{v_i \mid i \notin \{1,4\}\}$*
  3. *$V_3 = f(V_r \setminus V_l)$, % in the example on Fig. 5, $V_3 = \{w_6, w_7\}$*
  4. *V_4 consists of distinct copies of the nodes of the forest $\mathsf{query}_f(T)$, one for each node marked by \$ in right.*
     % in the example on Fig. 5, $V_4 = \{r_i \mid i \in \{6, \ldots, 10\}\}$
- *Setting $f(u) = u$ for all $u \in V_3$, we extend $f : V_r \cup V_l \rightarrow V_1 \uplus V_3$.*

- *$root' = f(root_r)$.*
- *Let $u \in V_1$ and let $\bar{u} = f^{-1}(u)$ be the associated node in $V_l \cap V_r$. If $\lambda_r(\bar{u}) \notin \mathcal{T}_{var}$ then $\lambda'(u) = \lambda_r(\bar{u})$, else $\lambda'(u) = t(\lambda_r(\bar{u}))$. For all $u \neq root'$, if $\mathsf{parent}_r(\bar{u})$ is defined, then $\mathsf{parent}'(u) = f(\mathsf{parent}_r(\bar{u}))$ else $\mathsf{parent}'(u) = \mathsf{parent}(u)$.*
- *For all $u \in V_2$, $\mathsf{parent}'(u) = \mathsf{parent}(u)$ and $\lambda'(u) = \lambda(u)$.*
- *For all $u \in V_3 \setminus \{root_r\}$, $\mathsf{parent}'(u) = f(\mathsf{parent}_r(u))$ and $\lambda'(u) = \lambda_r(u)$.*
- *To each node $u \in V'$ marked by \$, we add a copy of the forest $\mathsf{query}_f(T)$ as children of u, and we unmark the node.*

Note that if $x \in V_2$, then its parent is in $V_1 \cup V_2$. The same stands if $\mathsf{ancestor}_r(x)$ is defined. Note also that we indeed obtain a tree: for instance, if $u \in V_1$ and $\mathsf{parent}_r(\bar{u})$ is not defined, then $\mathsf{parent}(u)$ is defined. This is because $v = \mathsf{ancestor}_r(\bar{u})$ is then defined, so by Def. 3, $v = \mathsf{ancestor}_l(\bar{u})$ in left, so that $u = f(\bar{u})$ cannot be the root of T.

We write $T \xrightarrow{R} T'$ if T' can be obtained from T by applying the rule R. More generally, given a set of rules $\mathcal{R}$ we write $T \rightarrow T'$ if there is some rule $R \in \mathcal{R}$ with $T \xrightarrow{R} T'$, and $T \xrightarrow{*} T'$ for the reflexive-transitive closure of the previous relation. Notice that the tree T' matches right, through the matching $f' : V_r \rightarrow V'$ defined by $f'(v) = f(v)$ if $v \in V_r \cap V_l$, and $f'(v) = v$ if $v \in V_r \setminus V_l$.

Example (Play.com rules). To show how easily rules can be defined, we describe now some rules of the Play.com system. When the rule does not use a query or a guard, we only describe the left and right components.

- The rule New-Customer adds a new customer and its cart.
  - left = w_1[Play.com](w_2[MailOrder])
  - right =w_1[Play.com](w_2[MailOrder](w_3[Customer](w_4[Cart]))).
- Every new day, if a posted parcel has not yet been received yet, then the *day* counter is incremented.
  - left = w_1[Play.com](–w_2[Posted])
  - right = w_1[Play.com](–w_2[Posted](w_3[day]))
- If after 21 days a posted parcel is still not received, the customer can require a payback. We use the guard to ensure this time limit. Notice that the query is $Q \rightsquigarrow P$, where Q uses the same w_2 as in left, that is the number of days will be counted only for this particular parcel.
  - left = w_1[Play.com](–w_2[Posted])
  - Q= w_1[Play.com](–w_2[Posted](w_3[*day*])
  - P= x_4[*day*]
  - guard: a forest containing at least 21 trees (and possibly more nodes) whose root is labeled *day*.
  - right = w_1[Play.com].

3 Static Analysis of TPRS

We assume from now on that an active document is given by a *tree pattern rewriting systems (TPRS)* $(T, \mathcal{R})$, consisting of a set $\mathcal{R}$ of TP rules and a $\mathcal{T}$-labeled tree T. That is, we assume that each service corresponds to a rule. Our results are easily seen to hold in the more general setting where services are regular expressions over $\mathcal{R}$.

A tree T with node set V is *subsumed* by a tree T' with node set V', noted $T \preceq T'$, if there is an injective mapping from V to V' that preserves the labeling, the root, and the parent relation. A forest F is *subsumed* by a forest F', written $F \preceq F'$, if F is mapped injectively into F' such that each tree in F is subsumed by its image in F'. Similarly, a TP P with node set V is *subsumed* by a TP P' with node set V', if there is an injective mapping from V to V' that preserves the labeling, the root, the parent and the $\mathsf{ancestor}$ relations.

With a TPRS $(T, \mathcal{R})$ we can associate the (infinite-state) transition system $\langle S(T, \mathcal{R}), \rightarrow \rangle$ with $S(T, \mathcal{R}) = \{T' \mid T \xrightarrow{*} T'\}$. We are interested in checking the following properties:

- Termination: Are all derivation chains $T \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots$ of $(T, \mathcal{R})$ finite?
- Finite-state property: Is the set $S(T, \mathcal{R})$ of reachable trees finite?
- Reachability: Given $(T, \mathcal{R})$ and a tree T', is T' reachable in $(T, \mathcal{R})$?
- Confluence (joinability): For any pair of trees $T_1, T_2 \in S(T, \mathcal{R})$, does there exist some T' such that $T_1 \xrightarrow{*} T'$ and $T_2 \xrightarrow{*} T'$?
- Pattern reachability (coverability): Given $(T, \mathcal{R})$ and a tree pattern P, does $T \xrightarrow{*} T'$ hold for some T' matching P?
- Weak confluence: From any pair of trees $T_1, T_2 \in S(T, \mathcal{R})$, does there exist $T_1' \preceq T_2'$ such that $T_1 \xrightarrow{*} T_1'$ and $T_2 \xrightarrow{*} T_2'$?

For instance, pattern reachability is a key property when we want to talk about the reachability of some pattern. For example, we might ask whether an already cancelled order could still be delivered, which would mean a problem in the system. For this, it suffices to tag cancelled orders with a special symbol #, and check for the pattern $w_1[\text{Play.com}](-w_2[\text{delivered}](w_3[\#]))$. This is the same kind of properties which are checked in [3]. As expected, any of the non-trivial questions above is undecidable in the general case, see Theorem 1 below.

We are thus looking for relevant restrictions yielding decidability of at least some of these problems. In the next section we consider a subclass of TPRS, which is a special instance of the so-called *well-structured* systems. We say that $(T, \mathcal{R})$ is *positive* if all guards occurring in the rules from $\mathcal{R}$, are upward-closed. This means that for every guard G, and all forests F, F' with $F \preceq F'$, $F \in G$ implies $F' \in G$, too. In particular, if a rule R in a positive system is enabled for a tree T, then R is enabled for any tree T' that subsumes T. The reason is that for any TPQ query, we have that for every tree T_1' in $\mathsf{query}(T')$, there is some tree T_1 in $\mathsf{query}(T)$ such that T_1 is subsumed by T_1'. Notice that positive TPRS allow deletion of nodes, so they are more powerful than the positive AXML systems considered in [2].

The next theorem shows that upward-closed guards alone do not suffice for obtaining decidability of termination:

Theorem 1. *Any two-counter machine M can be simulated by a positive TPRS $(T, \mathcal{R})$ in such a way that M terminates iff $(T, \mathcal{R})$ terminates.*

Theorem 1 shows that any non trivial property is undecidable for positive TPRS without further restrictions. However, notice that the proof of the above result needs trees of unbounded depth. A realistic restriction in the XML setting is to consider only trees of bounded depth: XML documents are usually large, but shallow. A TPRS $(T, \mathcal{R})$ is called *depth-bounded*, if there exists some fixed integer K such that every tree T' with $T \xrightarrow{*} T'$ has depth at most K. Of course, Theorem 1 implies that it is undecidable to know whether a TPRS is depth-bounded. However, in many real-life examples this property is easily seen to hold (see *e.g.* the Play.com example, which has depth at most 8).

4 Decidability for positive and depth-bounded TPRS

For positive and depth-bounded TPRS, we can apply well-known techniques from the verification of infinite-state systems that are *well-structured*. Well-structured transition systems were considered independently by [1,8] and they cover many interesting models, such as Petri nets or lossy channel systems. We recall first some basics of well-structured systems.

Definition 5. *A well-quasi-order (wqo) on a set X is a preorder $\preceq$ such that in every infinite sequence $(x_n)_{n\geq 0} \subseteq X$, there exist some indices $i < j$ with $x_i \preceq x_j$.*

In general, the "subsumed" relation $\preceq$ on the set X of $\mathcal{T}$-labeled trees is not a wqo.[1] However, using Higman's lemma (see, *e.g.*, [6, Chap. 12]), one can show that $\preceq$ is a wqo on the set of trees of depth at most K (for any fixed K):

Proposition 1. *Fix $K \in \mathbb{N}$, and let X_K denote the set of unordered $\mathcal{T}$-labeled trees of depth at most K. The "subsumed" relation $\preceq \subseteq X_K \times X_K$ is a wqo.*

By the previous statement, a positive and depth-bounded TPRS $(T, \mathcal{R})$ yields a well-structured transition system $\langle S(T, \mathcal{R}), \rightarrow \rangle$ as defined in [8] (see also[2] [1]). This follows from the transition relation $\rightarrow$ being upward compatible: whenever $T \xrightarrow{R} T'$ and $T \preceq T_1$, there exists T_1' with $T_1 \xrightarrow{R} T_1'$ and $T' \preceq T_1'$.

For the next theorem we need first some notation. Given a set X and a preorder $\preceq$, we denote by $\uparrow X$ the upward closure $\{T' \mid T \preceq T' \text{ for some } T \in X\}$ of X. By $\min(X)$ we denote the set of minimal elements[3] of X. Finally, by $\mathrm{Pred}(X)$ we denote the set of immediate predecessors of elements of X. Note that whenever the transition relation $\rightarrow$ is upward compatible and X upward-closed, the set $\mathrm{Pred}(X)$ is upward-closed, too.

[1] Indeed consider the sequence of trees $(T_n)_{n\geq 0}$ where for each $n \geq 0$, T_n is the tree formed by a single branch of length $n+1$ whose internal nodes are labeled by a and the unique leaf is labeled by b.

[2] As shown in the proof of Thm. 2, $\langle S(T, \mathcal{R}), \rightarrow \rangle$ is also well-structured as defined in [1], which requires in addition that the set of predecessors of upward-closed sets is effectively computable.

[3] For a wqo $(X, \preceq)$ and $Y \subseteq X$, the set $\min(Y)/{\sim}$ is finite, where $\sim\, = \,\preceq \cap \preceq^{-1}$. For the subsumed relation $\preceq$, note that $\sim$ is the identity.

Since the subsumed relation $\preceq$ is a wqo, the $\preceq$ relation on forests is a wqo as well. Thus, each guard G in a positive, depth-bounded TPRS $(T, \mathcal{R})$ can be described by the (finite) set of forests $\min(G)$. The size $|G|$ of G is the maximal size of a forest in $\min(G)$.

Theorem 2. *Termination and pattern reachability are both decidable for positive and depth-bounded TPRS.*

Proof. First, termination is decidable for well-structured systems such that 1) $\preceq$ is decidable, 2) $\rightarrow$ is computable and 3) upward compatible, see [8, Thm. 4.6].

For pattern reachability, it is easy to see that the set of trees of depth bounded by some K which matches a TP P is upward-closed, and that the set of its minimal elements is effectively computable. We can thus use [1], which shows decidability of the reachability of $\uparrow T$ under the assumption that the set $\min(\mathrm{Pred}(\uparrow T))$ is computable. This allows to use the obvious backward exploration algorithm. From now on, we fix the tree T and the bound K of the system $(T_0, \mathcal{R})$. Let us also fix a rule $R = (\mathsf{left}, \mathsf{query}, \mathsf{guard}, \mathsf{right})$.

It suffices to consider the finite set $\mathcal{S}_R(T)$ of all trees T' of size at most $|T| + |\mathsf{left}| + K|\mathsf{query}||\mathsf{guard}|$ with $T' \xrightarrow{R} T$. Then, defining M as the set of minimal elements of $\bigcup_{R\in\mathcal{R}} \mathcal{S}_R(T)$, we get by definition $M \subseteq \min(\mathrm{Pred}(\uparrow T))$.

Let us now show that $M = \min(\mathrm{Pred}(\uparrow T))$. Let $T_1 \in \min(\mathrm{Pred}(\uparrow T))$. Thus, there exist some rule R and some injective matching (f, t) with $T_1 \xrightarrow{R} T$ via (f, t). Let also $F \in \min(\mathsf{guard})$ be a forest with $F \preceq F'$, where F' is the result of query on T_1 (compatible with the matching f).

The nodes of T_1 can be partitioned into 4 sets V_1, V_2, V'_3, V'_2 (below V_l are the nodes of left and V_r those of right): $V_1 = f(V_l \cap V_r)$, $V_2 = \mathsf{node}^{-1}(V_1)$, $V'_3 = f(V_l \setminus V_r)$, $V'_2 = \mathsf{node}^{-1}(V'_3)$. Notice that V_1 and V_2 are common with T (see Def. 4), hence $|T_1| \leq |T| + |V'_3| + |V'_2|$. Now, $|V'_3| \leq |\mathsf{left}|$. We now explain that V'_2 has at most $|\mathsf{query}||\mathsf{guard}|$ leaves, hence $|V'_2| \leq K|\mathsf{query}||\mathsf{guard}|$ which shows that $T_1 \in \mathcal{S}_R(T)$. Otherwise one can delete a leaf from V'_2 and get a tree $T'_1 \preceq T_1$ with $T'_1 \xrightarrow{R} T$ (via (f, t)), and still $F \preceq F''$, where F'' denotes the result of query on T'_1. This contradicts the minimality of T_1. □

On the negative side, depth-bounded well-structured systems can simulate reset Petri nets (i.e., nets with an additional transition that empties a place), hence we can deduce the following from known results:

Theorem 3. *Exact reachability, confluence, weak confluence and the finite-state property are undecidable for positive and depth-bounded TPRS.*

On the positive side, we can show that the finite-state property is decidable for positive, depth-bounded TPRS, that are *strict*, i.e., such that for any rule $(\mathsf{left}, \mathsf{query}, \mathsf{guard}, \mathsf{right})$, we require $V_{\mathsf{left}} \subseteq V_{\mathsf{right}}$. One cannot encode reset Petri nets with such systems because deletion is no longer possible (actually one can only relabel an existing node and create new nodes). Strict systems enjoy the additional property that whenever $T \overset{R}{\to} T'$ and $T \prec T_1$, there exists T'_1 with $T_1 \overset{R}{\to} T'_1$ and $T' \prec T'_1$ (notice that for non strict systems, we can only guarantee that $T' \preceq T'_1$). The results from [8] yield the following theorem.

Theorem 4. *The finite-state property and reachability are decidable for TPRS that are positive, depth-bounded, and strict.*

Note that the finite-state property is not very interesting in itself, but if it holds, then the other problems become decidable as we are dealing with a finite-state system. In particular, in order to test for confluence, it suffices to test that $(S(T,\mathcal{R}),\to)$ has a unique maximal strongly connected component.

Note that reachability is decidable for positive, depth-bounded and strict TPRS simply because $T \to T'$ implies that $|T| \leq |T'|$, and then it suffices to look for reachability of a tree T_1 in the finite state system $\{T' \mid |T'| \leq |T_1|\}$.

The table below sums up the results we obtained so far. It presents (un)decidability results concerning the various classes of positive TPRS we considered (depth-bounded and strict). The negative results about strict TPRS come from Theorem 1 (results on strict TPRS are obtained using slight variations of our proofs). Term., FS, Reach., P-reach, Confl. and W-confl. stand respectively for termination, finite state property, reachability, pattern reachability, confluence and weak confluence.

Model	Term.	FS	Reach.	P-reach.	Confl.	W-confl.
Strict	U	U	U	U	U	U
Depth-Bounded	D	U	U	D	U	U
Depth-Bounded & Strict	D	D	D	D	U	U

5 Lower bounds and extensions

Decidability results are obtained with non-constructive proofs coming from Higman's Lemma. This ensures termination of the algorithms, but without yielding complexity bounds. It is thus relevant to obtain lower bounds for these results.

Theorem 5. *The following problems have at least non-elementary complexity:*

- *Input: A TP P, a TPRS system $(S,\mathcal{R})$ and an integer k such that the depth of $(S,\mathcal{R})$ is bounded by k.*
- *Problem1: Is the pattern P reachable in $(S,\mathcal{R})$?*
- *Problem2: Does $(S,\mathcal{R})$ terminate,* equiv.*, does it have an infinite path?*

Proof. Let $\mathrm{tower}(0,n) = n$ and $\mathrm{tower}(k+1,n) = 2^{\mathrm{tower}(k,n)}$. Let M be an $n \mapsto \mathrm{tower}(k,n)$-space bounded deterministic Turing machine and x be an input of M. Denote by $\log^* n$ the smallest integer m such that $n \leq \mathrm{tower}(m,2)$ and let $K = k + \log^* |x|$, so that the computation of M on x uses at most $\mathrm{tower}(k,|x|) \leq \mathrm{tower}(K,2)$ tape cells. We build a $(K+1)$-depth bounded TPRS of size $O(|M|+|x|)$ simulating M on x.

Informally, we encode each configuration of M by a tree. Each cell is encoded by a subtree of the root, labeled at its own root by the cell content, with the forest below it encoding the position of the cell. Since such a position is smaller than tower$(K, 2)$, it can itself be encoded recursively by a forest of depth at most K (such a recursive encoding of large integers, by words, has already been used in [15]). For instance, one can encode integers from 0 to $15 = \text{tower}(\mathbf{2}, 2) - 1$ at depth **2**. The forest of Fig. 6 encodes 13 (**1101** in binary). To recover its position, each bit of the base 2 representation has under itself a forest of depth 1 encoding its position (recursively with the same encoding scheme). For instance, the leftmost **1** is at position 00, which is encoded by the forest $\{[0]([0]), [0]([1])\}$.

Let $N = \text{tower}(K, 2) - 1$. We encode the configuration C of M with tape content $a_0 \cdots a_N$, current state q and scanned position m, by the forest $F_C = \bar{a}_0(F_0^K) \cdots \bar{a}_N(F_N^K)$ of depth $K + 1$, with $\bar{a}_m = [a_m, q]$ and $\bar{a}_i = [a_i]$ for $i \neq m$, where F_i^K is the forest of depth K encoding the number $i \leq N$. The head position is thus doubly tagged: by the letter, and by the state. Such a node with a double tag $[\alpha, \beta]$ is said *marked by* β, or a β*-node*.

In order to navigate through the cells, we use for each level $\ell \leq K$ an additional placeholder node, child of the root, named c_ℓ for holding a level ℓ counter below it. The idea is that the counter attached below c_K will be able to count up to N, and hence can pinpoint a tape position. The other counters are needed in the inductive process. During the computation, additional markers will be used either as pebbles, or to guide the control. Figure 7 shows a typical tree reached during the computation. The rules of the TPRS are set up so that it performs successively the following actions:

1. It creates the forest F_{C_0}, and attaches it under the root, leaving c_K labeled by $[c_K, \mathsf{run}]$ and for $\ell < K$, c_ℓ labeled by $[c_\ell, \mathsf{end}]$.
2. It simulates repeatedly transitions of M, stopping if the final state is reached.

We only show how to encode transitions. The generation of the initial configuration, starting from the tree $[M]([c_0, \mathsf{end}], \ldots, [c_{K-1}, \mathsf{end}], [c_K, \mathsf{create\text{-}init\text{-}config}])$, is done using similar routines. We use a finite set of rules without query/guard part. Although the TPRS will be nondeterministic, appropriate tags shall ensure that rules applicable at some step have all the same *left* member. When the TPRS discovers that a nondeterministic guess was wrong, it blocks. Therefore, if M halts on x, then the TPRS always terminate. If M does not halt on x, then the corresponding run of the TPRS where all guesses are correct does not either. This ensures termination iff M halts on x.

To simulate a transition, the TPRS first performs the changes in the configuration, nondeterministically guessing the new head position. To check whether the head has been properly placed, it $*$-marks the original head position. The node c_K marked by $\{$ run, $\mathsf{check\text{-}suc1}$, $\mathsf{check\text{-}suc2}$, $\ldots$, $\mathsf{check\text{-}pred1}$,$\mathsf{check\text{-}pred2}\ldots\}$

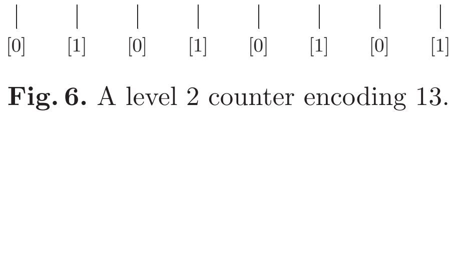

Fig. 6. A level 2 counter encoding 13.

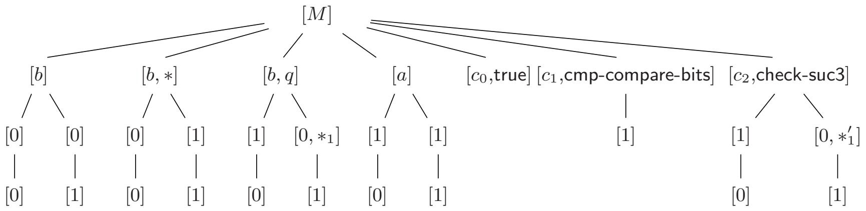

Fig. 7. The tree coding the tape $b\,b\,q\,b\,a$ of the Turing machine M.

(for instance, check-suc needs several steps) encodes the current stage of the simulation. For instance, to simulate a transition $p \xrightarrow{a/b/\rightarrow} q$, we use the rule:

- left $= r[M](x[a, p],\ y[X],\quad z[c_K, \mathsf{run}])$,
- right $= r[M](x[b, *],\ y[X, q],\ z[c_K, \mathsf{check\text{-}suc1}])$,

slightly abusing notation: we use a double tag involving a variable, to abbreviate a finite set of rules (with the obvious interpretation).

To complete the simulation of the transition, the TPRS checks whether the position written below the node pinpointed by q is a successor of that below the node pinpointed by $*$. If yes, it deletes the mark $*$, and labels c_K back to $[c_K, \mathsf{run}]$. If not, the head position was incorrectly guessed and the system blocks.

The steps of check-suc, which checks that the nodes marked by $*$ and q occur successively, are first a copy under c_K of the level K counter located below the $*$-node, then an increment of that copy, and a comparison of the result to the counter below the q-node. We use auxiliary markers $*_\ell$, $*'_\ell$ for each level ℓ, attached to nodes below an ℓ counter: $*_\ell$ in the part of the tree representing the configuration, and $*'_\ell$ under some c_i, $i > \ell$. We define inductively rules to achieve the following tasks for each level $\ell \leq K$:

- copy(ℓ) copies below the $*'_\ell$-marked node the level ℓ counters found below c_ℓ.
- increment(ℓ) increments the level ℓ counter below c_ℓ.
- compare(ℓ) compares level ℓ counters below c_ℓ and below the $*_\ell$-marked node.
- test-max(ℓ) tests if the level ℓ counter below c_ℓ has its maximal value.
- zero(ℓ) generates under c_ℓ the level ℓ counter F_0^ℓ.

Each task of level ℓ is implemented by a sequence of tasks of level $(\ell - 1)$, using some fresh tags to correctly organize the order of these level $(\ell - 1)$ tasks. See [9] for some detailed rules. □

The bounded depth restriction needed for our decidability results can be relaxed if we forbid the use of the direct parent-child edges in tree patterns. This leads to the following preorder on unranked, unordered $\mathcal{T}$-labeled trees, which is a well quasi-order by Kruskal's theorem (see [6, Chap. 12]). For two trees T, T' with sets of nodes V, V' respectively, we write $T \lessdot T'$, if there is an injective mapping from V to V' that preserves the labeling, the root, and the ancestor relation. So compared to the relation $\preceq$ used previously, we do not require that the parent relation is preserved.

Clearly, we need to restrict the TPRS rules in order to obtain well-structured systems. Namely we require that all TP occurring in query and left use only ancestor edges (right can still use parent edges, but the parent relation cannot be tested for). We call such TPRS *undirected*. Using similar proofs as in Sect. 4, we get the same decidability results. For the lower bound we obtain a stronger result, by encoding reachability for lossy channel systems (LCS). These are finite-state machines communicating over FIFO channels that can loose arbitrary many messages. Reachability for LCS has non primitive recursive complexity [14], already for LCSs made up of two finite-state machines and two channels [5].

Theorem 6. *Termination and pattern reachability have at least non primitive recursive complexity for undirected TPRS.*

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS'96*, pp. 313–321. IEEE Comp. Soc., 1996.
2. S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *PODS'04*, pp. 35–45. ACM, 2004.
3. S. Abiteboul, L. Segoufin, and V. Vianu. Static Analysis of Active XML Services. In *PODS'08*. ACM, 2008. To appear.
4. Active XML. `http://www.activexml.net/`.
5. P. Chambart and Ph. Schnoebelen. The Ordinal Recursive Complexity of Lossy Channel Systems. In *LICS'08*. IEEE Comp. Soc., 2008. To appear.
6. R. Diestel. *Graph theory.* 2005. `http://www.math.uni-hamburg.de/home/diestel`.
7. C. Dufourd, A. Finkel and Ph. Schnoebelen. Reset Nets between Decidability and Undecidability. In *ICALP'98*, LNCS 1443, pp. 103–115. Springer, 1998.
8. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
9. B. Genest, A. Muscholl, O. Serre, M. Zeitoun. Tree Pattern Rewrite Systems. Internal report available at `http://www.crans.org/~genest/GMSZ08.pdf`.
10. N. Dershowitz and D. Plaisted. *Chapter 9 in: Handbook of Automated Reasoning, vol. 1*, A. Robinson and A. Voronkov eds, Elsevier (2001).
11. L. Libkin. Logics over unranked trees: an overview. *Logical Methods in Computer Science*, 2(3), 2006.
12. Ch. Löding and A. Spelten. Transition Graphs of Rewriting Systems over Unranked Trees. In *MFCS'07*, LNCS 4708, pp. 67-77. Springer, 2007.
13. F. Neven. Automata, Logic, and XML. In *CSL'02*, LNCS 2471, pp. 2–26. Springer, 2002.
14. Ph. Schnoebelen. Verifying Lossy Channel Systems has Nonprimitive Recursive Complexity. *Inf. Process. Lett.* 83(5):251–261, 2002.
15. I. Walukiewicz. Difficult Configurations—on the Complexity of LTrL. *Form. Methods Syst. Des.*, 26(1): 27–43. Kluwer, 2005. Short version in *ICALP'98*, LNCS 1443.