

Verifying Recursive Active Documents with Positive Data Tree Rewriting

Blaise Genest
CNRS, IRISA, Rennes, France

Anca Muscholl
LaBRI, Université Bordeaux 1,
France

Zhilin Wu
LaBRI, Université Bordeaux 1,
France

ABSTRACT

This paper proposes a data tree-rewriting framework for modeling evolving documents. The framework is close to Guarded Active XML, a platform used for handling XML repositories evolving through web services. We focus on automatic verification of properties of evolving documents that can contain data from an infinite domain. We establish the boundaries of decidability, and show that verification of a *positive* fragment that can handle recursive service calls is decidable. We also consider bounded model-checking in our data tree-rewriting framework and show that it is *NexpTime*-complete.

Keywords

Active documents, Guarded Active XML, automated verification, data tree rewriting, tree decompositions.

1. INTRODUCTION

From static in house solutions, databases have become more and more open to the world, offering e.g. half-open access through web services. As usual for open systems, their design requires a careful static analysis process, helping to guarantee that no malicious client may take advantage of the system in a way for which the system was not designed. Static analysis of such systems very recently brought together two areas - databases, with emphasis on semi-structured XML data, and automated verification, with emphasis on model-checking infinite-state systems. Systems modeling dynamical evolution of data are pretty challenging for automated verification, as they involve feedback loops between semi-structured data, possibly with values from unbounded domains, and the workflow of services. If each of these topics has been studied extensively on its own, very few papers tackle decidability of algorithms when all aspects are present at the same time.

An interesting platform emerged recently for using XML repositories evolving through web services, namely Active XML (AXML) [4]. These are XML-based documents that

evolve dynamically, containing implicit data in form of embedded service calls. Services may be recursive, so the evolution of such documents is both non-deterministic and unbounded in time. A first paper analyzing the evolution of AXML documents considered *monotonous* documents [3]. With this restriction, as soon as a service is enabled in an AXML tree T , then from this point on the service cannot be disabled, and calling it can only extend T . In particular, information cannot be deleted and subsequent service calls return answers that extend previous answers. Recently, a workflow-oriented version of AXML was proposed in [5]: the *Guarded AXML* model (GAXML for short) adds guards to service calls, thus controlling the possible evolution of active documents. Decidability in *co-2NexpTime* of static analysis for *recursion-free* GAXML w.r.t Tree-LTL properties was established in [5]. The crucial restriction needed for decidability there is a uniform bound on the number of possible service calls. Compared to [3], service invocation can terminate, and more importantly, negative guards can be used. Even more importantly, verification tasks are more complex and challenging because of the presence of unbounded data. However, the model relies on a rigid semantics of what a service call can do, and how (using a workspace etc). For instance, deletion of data is not possible.

In this work, our aim is twofold. First, we aim at extending the GAXML model in a uniform way, by expressing the effect of embedded service calls in form of tree rewriting rules. Our model DTPRS (*data tree pattern rewriting systems*) is based on the same basic ingredients as GAXML, which are tree patterns for guards and queries. However, our formalism allows a user to describe several possible effects of a service call: materialization of implicit data like in (G)AXML, but also deletion and modification of existing document parts. This model is a simplified version of the TPRS model proposed in [15], but it can additionally handle data from infinite domains.

Our second, and main objective is to get decidability of static analysis of DTPRS without relying on a bound on the number of service calls. For doing that, we use a technique that emerged in the verification of particular infinite-state systems, such as Petri nets and lossy channel systems. The main concept is known in verification as *well-structured transition systems* (WSTS for short) [1, 13]. WSTS are one example of infinite-state systems where (potentially) infinite sets of states can be represented (and effectively manipulated) symbolically in a finite way. In contrast, [5] uses a small model property which implies an enumeration of trees up to some bound.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

Our basic objects are data trees, i.e., trees with labels from an infinite domain. We view data trees as graphs, and define in a natural way a well-quasi-order on such graphs. Then we show that a uniform bound on the length of simple paths in such graphs, together with positive guards, makes DTPRS well-structured systems [1, 13]. As a technical tool we use here tree decompositions of graphs. In a nutshell we trade here recursion against positiveness, since considering both leads to undecidable static analysis. We show that for *positive* DTPRS, termination and tree pattern reachability are both decidable. Furthermore, we show that *bounded* model-checking of (*not necessarily positive*) DTPRS is **Nexptime**-complete. On the other hand, we show that the verification of simple but non positive temporal properties is undecidable even for *positive* DTPRS.

Related work: Verification of web services often ignores unbounded data (c.f. e.g. [17, 14]). On the other hand, several data-driven workflow process models have been proposed. Document-driven workflow was proposed in [20]. Artifact-based workflow was outlined in [16], in which artifacts are used to represent key business entities, including both their data and life cycles. An early line of results involving data establishes decidability boundaries for the verification of temporal (first-order based) properties of a data-driven workflow processes, based on a relational data model [11, 10, 12]. This approach has been recently extended to the artifact-based model [9].

On the verification side, there is a rich literature on the verification of well-structured infinite transition systems [1, 13], ranging from faulty communication systems [7] to programs manipulating dynamic data [2] (citing only a few recent contributions). The latter work is one of the few examples where well-quasi-order on graphs are used.

Organization of the paper: In the next section, we fix some definitions and notations, define the DTPRS model, and illustrate how to reduce GAXML to our DTPRS model. Then in Section 3, we describe an example to illustrate the expressivity of DTPRS. In Section 4, we show that DTPRS with recursive DTD or negated tree patterns are undecidable. In Section 5 we define positive DTPRS and prove our decidability results. On the other hand, we show the undecidability of the verification of general, non-positive temporal properties in Section 6. Finally in Section 7, we consider bounded model-checking of (*not necessarily positive*) DTPRS and show that the bounded model-checking problem is **Nexptime**-complete. Omitted proofs can be found in the appendix.

2. DEFINITIONS AND NOTATIONS

In this paper, documents correspond to labeled, unranked and unordered trees. Fix a finite alphabet Σ (with symbols a, b, c, \dots , called *tags*) and an infinite data domain \mathcal{D} . A *data tree* is a (rooted) tree T with nodes labeled by $\Sigma \cup \mathcal{D}$. A data tree T can be represented as a tuple $T = (V, E, \text{root}, \ell)$, with labeling function $\ell : V \rightarrow \Sigma \cup \mathcal{D}$. Inner nodes are Σ -labeled, whereas leaves are $(\Sigma \cup \mathcal{D})$ -labeled. We fix a finite set of variables \mathcal{X} (with symbols X, Y, Z, \dots) that will take values in \mathcal{D} , and use $*$ as special symbol standing for any tag. Let \mathcal{T} denote $\Sigma \cup \mathcal{X} \cup \{*\}$.

A *data constraint* is a Boolean combination of relations $X = Y$, with¹ $X, Y \in \mathcal{X}$.

¹For simplicity we disallow here explicit data constants $X =$

A *data tree pattern* (DTP) $P = (V, E, \text{root}, \ell, \tau, \text{cond})$ is a (rooted) \mathcal{T} -labeled tree, together with an edge-labeling function $\tau : E \rightarrow \{ |, || \}$ and a data constraint *cond*. As usual, $|$ -labeled edges denote child edges, and $||$ -labeled edges denote descendant edges. Internal nodes are labeled by $\Sigma \cup \{*\}$, and leaves by \mathcal{T} . A *matching* of a DTP P into a data tree T is defined as a mapping preserving the root, the Σ - and \mathcal{D} -labels (with $*$ as wildcard), the child- and the descendant relations, satisfying *cond* and mapping \mathcal{X} -labeled nodes to \mathcal{D} -labeled ones. In particular, a relation $X = Y$ ($X, Y \in \mathcal{X}$) means that the corresponding leaves are mapped to leaves of T carrying the same data value. If the mapping above is injective, then it is called an *injective* matching of P into T .

A *relative* DTP is a DTP with one designated node *self*. A relative DTP (P, self) is matched to a pair (T, v) , where T is a tree and v is a node of T .

We consider *Boolean combinations* of (relative) DTPs. The patterns therein are matched *independently* of each other (except that nodes designated by *self* must be matched to the same node of T), and the Boolean operators are interpreted with the standard meaning.

A *data tree pattern query* (DTPQ) is of the form *body* \rightsquigarrow *head*, with *body* a DTP and *head* a tree such that

- the internal nodes of *head* are labeled by Σ and its leaves are labeled by $(\Sigma \cup \mathcal{D} \cup \mathcal{X})$,
- every variable occurring in *head* also occurs in *body*,
- there is at least one variable occurring in *head*, i.e., at least one leaf of *head* is labeled by \mathcal{X} .

Let T be a data tree and $Q = \text{body} \rightsquigarrow \text{head}$ be a DTPQ. The evaluation result of Q over T is the forest $Q(T)$ of all instantiations of *head* by matchings μ from *body* to T . A *relative* DTPQ is like a DTPQ, except that its *body* is a relative pattern.

A *locator* is a relative DTP L with additional labels from the set $\{\text{append}, \text{del}\} \cup \{\text{ren}_a \mid a \in \Sigma\}$. The labels *append* and *ren_a* are not exclusive and can be attached only to nodes of L that are labeled by a tag (that is by $\Sigma \cup \{*\}$) but not by $\mathcal{D} \cup \mathcal{X}$. Nodes not labeled by *append*, *ren_a* can be labeled by *del* (even data nodes), such that the descendants of a node labeled by *del* are labeled by *del*, too. The intuition behind this definition is to provide a context for data tree rewriting rules, together with some possible actions on this context: renaming, deletion, appending.

A *data tree rewriting rule* (DTP rule) R is a tuple $(L, G, \mathcal{Q}, \mathcal{F}, \chi)$ with:

- L is a *locator*,
- G is a Boolean combination of (relative) DTPs (the *guard* of R),
- \mathcal{Q} is a finite set of relative DTPQs,
- \mathcal{F} is a finite set of forests with internal nodes labeled by Σ and leaves labeled by $\Sigma \cup \mathcal{D} \cup \mathcal{X} \cup \mathcal{Q}$,
- χ is a mapping from the set of nodes of L labeled by *append* to \mathcal{F} .

A *DTP rewriting system* (DTPRS) is a pair (\mathcal{R}, Δ) consisting of a set \mathcal{R} of DTP rules and a *static invariant* Δ , d ($d \in \mathcal{D}$): they can be simulated by tags from Σ .

consisting of a DTD and a *data invariant*, i.e. a Boolean combination of DTPs. We assume that the static invariant Δ is preserved by the rewriting rules \mathcal{R} . As usual for unordered trees, a DTD is defined as a tuple (Σ_r, \mathcal{P}) such that Σ_r is the set of allowed root labels, and \mathcal{P} is a finite set of rules $a \rightarrow \psi$ such that $a \in \Sigma$ and ψ is a Boolean combination of inequalities of the form $|b| \geq k$, where $b \in \Sigma \cup \{dom\}$ (dom is a symbol standing for any data value), and k is a non-negative integer. A *positive DTD* is one where the Boolean combinations above are positive.

We now define formally the semantics of a transition by DTP rules. So let $T = (V, E, \text{root}, \ell)$ be a data tree (with $T \models \Delta$) and let $R = (L, G, \mathcal{Q}, \mathcal{F}, \chi)$ be a DTP rule.

- Let μ be an *injective* matching from L to T . Let ν be the assignment of data values to variables in L such that $\nu(X) = \ell(\mu(v))$ for every v labeled by $X \in \mathcal{X}$ in L .
- For each variable $X \in \mathcal{X}$ we denote its evaluation as $X(T)$, with $X(T) = \nu(X)$ if defined, and $X(T)$ a **fresh data value otherwise**. Here a fresh data value is a data value which does not appear anywhere else in T . Furthermore, it is required that all the *new* variables of R , i.e. variables occurring in \mathcal{F} , but not in L , should take **mutually distinct** fresh values. For each forest $F \in \mathcal{F}$, we denote its evaluation by $F(T)$, by replacing labels $Q \in \mathcal{Q}$ by $Q(T)$ and labels $X \in \mathcal{X}$ by $X(T)$. Recall that all queries $Q \in \mathcal{Q}$ are evaluated relatively to $\mu(\text{self})$.
- A data tree T' can be obtained from T by
 - deleting subtrees rooted at nodes $\mu(v)$ whenever v is labeled by *del* in L ,
 - changing the tag of a node $\mu(v)$ to a whenever v is labeled by *ren_a* in L ,
 - appending $F(T)$ as a subforest of nodes $\mu(v)$ whenever v is labeled by *append* in L and $\chi(v) = F$,
 - every other node of T keeps its tag or data.
- The rule R is enabled on data tree T if there exists an *injective* matching μ of L into T such that (1) G is true on $(T, \mu(v))$ with v labeled by *self* in L , and (2) there is a data tree T' , obtained from T and μ by the operations specified above, satisfying $T' \models \Delta$.

Let $T \xrightarrow{R} T'$ denote the transition from T to T' using DTP rule $R \in \mathcal{R}$.

REMARK 1. 1. *The injectivity of the matching μ ensures that the outcome T' is well-defined. In particular, no two nodes with label *del* and *append* (or *ren_a*), resp., can be mapped to the same node in the data tree. However, mappings used for guards or queries are - as usual - non injective.*

2. *For the new variables occurring in \mathcal{F} , but not in L , we choose mutually distinct fresh values. We could have chosen arbitrary values instead, and enforce the fact that they are fresh and mutually distinct a posteriori using the invariant Δ . In this case, the invariant needs negation. The invariant (or the locator) can be also used to enforce that the (arbitrarily) chosen values already occur in T . This kind of invariant would be positive.*

3. *In our definition of DTP rules, it might appear that guards are redundant wrt. the locator. But notice that this only concerns positive guards.*

Given a DTPRS (\mathcal{R}, Δ) , let $T \longrightarrow T'$ denote the union of $T \xrightarrow{R} T'$ for $R \in \mathcal{R}$, and $T \xrightarrow{+} T'$ (or $T \xrightarrow{*} T'$) denote the transitive (or reflexive and transitive) closure of $T \longrightarrow T'$. Moreover, let $\mathcal{T}_{\mathcal{R}}^*(T)$ denote the set of trees that can be reached from a data tree T by rewriting with DTP rules from \mathcal{R} , i.e. $\mathcal{T}_{\mathcal{R}}^*(T) = \{T' \mid T \xrightarrow{*} T'\}$. For a set of data trees \mathcal{I} , let $\mathcal{T}_{\mathcal{R}}^*(\mathcal{I})$ be the union of $\mathcal{T}_{\mathcal{R}}^*(T)$, for $T \in \mathcal{I}$.

We are interested in the following questions, given a DTPRS (\mathcal{R}, Δ) :

- *Pattern reachability:* Given a DTP P and a set of initial trees² Init , given as the conjunction of a DTD and a Boolean combination of DTPs, is there some $T \in \mathcal{T}_{\mathcal{R}}^*(\text{Init})$ such that P matches T ?
- *Termination:* Given an initial data tree T_0 , are all rewriting paths $T_0 \rightarrow T_1 \rightarrow \dots$ starting from T_0 finite?

The reason for the fact that termination of DTPRS is defined above w.r.t a single initial data tree is that termination from a set of initial trees is already undecidable without data (see Section 4).

2.1 Reduction from GAXML to DTPRS

DTPRS is a quite powerful model, which allows to model e.g. Guarded Active XML (GAXML) [5]. We show this translation on two main GAXML steps: call and return of services. For completeness, we briefly recall the main features of GAXML here. AXML trees contain embedded function calls that are evaluated (via tree pattern queries) in a workspace. GAXML adds call and return guards, that control the function call and the return of the result (as sibling of the call node). Functions can be internal or external. The external ones return some arbitrary forest that is consistent with the static invariant Δ .

We describe here how to model an (internal) function call in GAXML with DTPRS. Let $f \in \Sigma$ be a function associated with the argument query Q and the call guard G . The associated DTP rule has the same guard G , the set of queries $\mathcal{Q} = \{Q\}$, and $\mathcal{F} = \{T_f, T_X\}$ is the set defined below:

- T_f is a tree with three nodes, the root being labeled by f , and the two leaves labeled by the query Q and by the variable X , respectively.
- T_X is a tree with a unique node labeled by the variable X ,

The locator L is given in Figure 1. Finally, χ maps the $!f$ -node to T_X and the WS-node to T_f . Applying the DTP rule amounts in evaluating Q to get the arguments of the call, writing them into the workspace WS, and creating a fresh identifier X that is copied both below WS and below the node with the function call (aka return address for f , see below).

We describe now how to model the return of an (internal continuous) function $f \in \Sigma$ associated with the return query Q and the return guard G . The associated DTP rule has the same guard G , the set of queries $\mathcal{Q} = \{Q\}$, $\mathcal{F} = \{T_Q\}$ has

²We require that every tree in Init satisfies Δ .

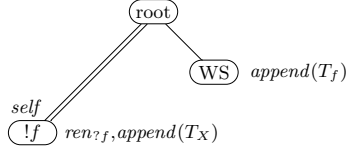


Figure 1: The locator for service invocation.

a unique tree with a unique node labeled by the query Q , and the locator is given in Figure 2 with the data constraint $X = Y$:

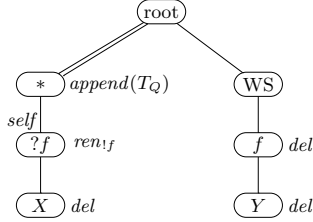


Figure 2: The locator for service return.

Finally, χ maps the node labeled *append* with T_Q . This DTP rule locate where the call was computed in the workspace WS using X , evaluates Q to get the return of the call, puts the result of the return query as a sibling of $?f$, and deletes the associated data in the workspace, as well as the identifier X .

3. MAILORDER EXAMPLE

The following is a DTPRS description of the basic functionalities of a MailOrder system for *Play.com*. For simplicity, we represent only what happens on the *Play.com* peer, although we could also model client peers, bank peers etc.

The *Play.com* example can be compared with the *MailOrder* example in [5]. Syntactically, GAXML uses guards and queries. Most of the time, guards and queries are very simple and can be encoded in the locator of DTP rules. In this case, we omit the *self* label in our rules. Unlike *MailOrder* we can express deletion with DTPRS, and thus model the selection of the products in the cart (adding and deleting products), and also handle an inventory (how many items of a product remain - each time an item is added to a cart, it is also deleted from the inventory). More importantly, compared with the recursion-free decidable restriction of GAXML, we are able to represent the process of many customers ordering many different products in our decidable fragment.

On the *Play.com* peer, there are a product catalog, a customer catalog, a set of carts and a set of orders. The inventory is encoded in the product catalog: if there are three items of a product in the inventory, then there are three tokens as children of the product. Each cart is associated with a customer (at first anonymous, and he can later login in as a registered member). The cart is first in the select mode, which allows the associated customer to add/delete products. Then the customer can check out, the cart gathers the different prices for the products into a bill, and goes into payment mode. When the customer pays, a corresponding order is created with a receipt, the customer is disconnected and the cart is deleted.

A simple example of a configuration of *Play.com* is illus-

trated in Figure 3. We represented a data value only when it is used by at least two different nodes.

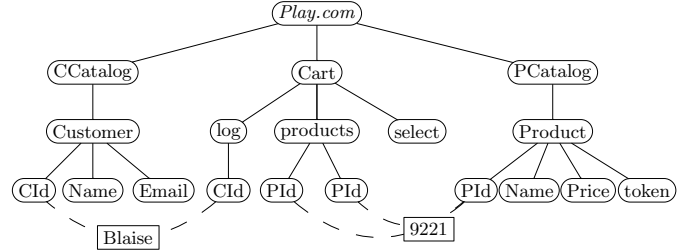


Figure 3: A configuration of the *Play.com* system, where the registered customer Blaise has twice the same product in his cart, with PID 9221. There is one additional product with PID 9221 left.

Some key rewriting rules are described in the following. We only describe nontrivial components in these rules. Tree patterns are represented below in term form, with descendant edges preceded by a $-$ symbol.

- An anonymous customer can connect to *Play.com* with the rule *create-cart*.
 - $- L := [Play.com]_{append(F)}$
 - $- F := [Cart]([nolog]([CId](X)), [products], [select])$. Notice that here X is a fresh data value.
- An anonymous customer can login as a registered member with the rule *login*. As the customer peer is not modeled, we do not handle the check of a password, although it would not be a problem to do so in our framework.
 - $- L := [Play.com](-[Customer]([CId](X)), [Cart]_{append(F)}([nolog_{del}]))$
 - $- F := [log]([CId](X))$. Here, X is the same PID for the cart and for the customer.
- The rule *Add-Product* adds a new product into the cart (and deletes a token from the inventory).
 - $- L := [Play.com]([Cart]([products]_{append(F)}), [Product]([PId](X), [token_{del}]))$
 - $- F := [PId](X)$.
- The rule *Delete-Product* deletes a product from the cart (and puts the token back).
 - $- L := [Play.com]([Cart]([select], -[PId_{del}](X)), [Product]_{append(F)}([PId](X)))$
 - $- F := [token]$.
- The rule *Check-out* checks whether the cart is nonempty and retrieves the prices of products in the cart into a bill through a query. It changes the mode of the cart from *select* to *payment*.
 - $- L := [Play.com]([Cart]_{self, append(F)}(-[PId], [select]_{renpayment}))$

- $F := [\text{Bill}](Q)$, and $Q := \text{body} \rightsquigarrow Y$ is the query with $\text{body} = [\text{Play.com}](\text{[Cart}_{\text{self}}](\text{--[Pid]}(X)), \text{--[Product]}(\text{[Pid]}(X), \text{[Price]}(Y)))$.

- The customer can pay with the rule *Pay*, and a corresponding order is created. For simplicity, it disconnects the customer, and transforms the cart into an order. The order contains the customer ID, an order ID (a fresh unique identifier), and the total price (sum of the prices of each items). As we model prices by data values and we do not use any arithmetics, the total price is a fresh data value. The only important thing is that this data value *Total* is the same as the one registered in the bank account, which we could check for equality (although we do not explicitly model the bank here). The order does not recall the individual PID of products since there will be no more products to put back in the inventory (and anyway the product can be later removed from the catalog).

- $L := [\text{Play.com}](\text{[Cart}_{\text{append}_F, \text{ren}_{\text{Order}}}](\text{[Bill}_{\text{del}}], \text{[products}_{\text{del}}], \text{[payment}_{\text{ren}_{\text{paid}}}], \text{[log}_{\text{ren}_{\text{cust}}}]$
- $F := [\text{Receipt}](\text{[OID]}(X), \text{[Total]}(Y))$

- The rule *Add-member* allows a customer to register as a member.

- $L := [\text{Play.com}](\text{[CCatalog}_{\text{append}}(F)])$
- $F := [\text{Customer}](\text{[CID]}(X), \text{[Name]}(Y), \text{[Email]}(Z))$

We do not specify the following rules here, which are easy to come up with: *shipped*, *delivered*, *add product to catalog* etc.

Notice that this example is intentionally not correct. Indeed, as there is no check of the state in which the cart is at rule *Add-Product*, it is possible that a bill is produced after a check out, and the customer can still add products with the aforementioned rule which will never be accounted for in the bill. We will show later that we can decide whether such a problem occurs or not. To fix this problem, it suffices to check in the locator L of *Add-Product* rule that the cart is in the select mode.

4. UNDECIDABILITY

As one might expect, analysis of DTPRS is quickly undecidable – and sometimes already without using any unbounded data. The proof of the proposition below is obtained by straightforward simulations of 2-counter machines.

PROPOSITION 2. *Both termination and pattern reachability for DTPRS (\mathcal{R}, Δ) are undecidable whenever one of the following holds:*

1. *the DTD in Δ is recursive,*
2. *either guards in \mathcal{R} or the invariant Δ contain negated DTPs.*

The above result holds even without data.

The next result shows that with data, we can relax both conditions above and still get undecidability of DTPRS. The main idea is to use data for creating long horizontal paths (although trees are supposed to be unordered). Such horizontal paths can be obtained e.g. with a tree of depth 2,

having subtrees of height one with 2 leaves each, say labeled by data values d_i, d_{i+1} . Assuming all d_i are distinct (and distinguishing d_1) we get a linear order on these subtrees.

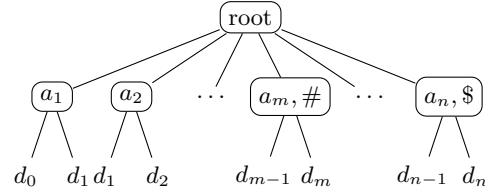
THEOREM 3. *Both termination and pattern reachability are undecidable for DTPRS (\mathcal{R}, Δ) such that (1) the DTD in Δ is non-recursive and (2) all DTPs in guards from \mathcal{R} and the invariant Δ are positive.*

PROOF. We reduce Post correspondence problem (PCP) first to pattern reachability. We may assume that our PCP instance $(u_i, v_i)_{1 \leq i \leq n}$ is such that the following holds for every non-empty sequence i_1, \dots, i_k of indices:

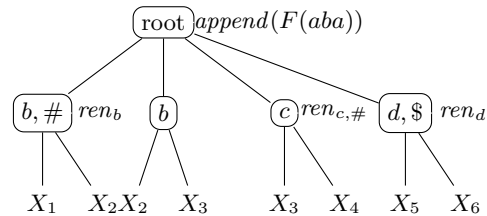
- Either $U = u_{i_1} \dots u_{i_k}$, $V = v_{i_1} \dots v_{i_k}$ are incomparable, or V is a prefix of U . In the latter case we call (U, V) a partial solution.
- If (U, V) and (Uu_i, Vv_i) are partial solutions and $U \neq V$, then either $Uu_i = Vv_i$ or Vv_i is a prefix of U .
- Every solution starts with the pair (u_1, v_1) and ends with (u_n, v_n) .

It is not hard to verify that the usual Turing machine reduction to PCP satisfies the restrictions above.

A partial solution (U, V) with $U = a_1 \dots a_n$, $V = a_1 \dots a_{m-1}$ will be represented by the data tree below. In this data tree the leaves are labeled by data d_i , with $d_i \neq d_j$ for all $i \neq j$. Moreover, notice that the last position has the special marker \$, and the first position in U without V has the special marker #.



With each PCP pair (u_i, v_i) , $i < n$, we associate DTPRS rules $R_i = (L, G, Q, \mathcal{F}, \chi)$. For simplicity we describe below the locator L and the forest \mathcal{F} for $(u_i, v_i) = (aba, bb)$ (the guard G and the query set Q are both empty):



We need a rule R_i for each pair of tags $c, d \in \Sigma$ (these are the tags at positions n and $m+2$ in the example with $|v_i| = 2$). The forest $F(aba)$ which will be added under the root node will contain 3 trees T_1, T_2, T_3 , with roots labeled a, b and $a, \$$, respectively. Tree T_1 has two leaves, labeled X_6 and X_7 . Tree T_2 (T_3 , resp.) has two leaves, labeled X_7 and X_8 (X_8 and X_9 , resp.). Notice that variable X_6 occurs in both L and $F(aba)$, whereas X_7, X_8, X_9 will take fresh (and mutually distinct) values.

The pair (u_n, v_n) has similar rules, except that we will not append any forest to the root, but rename the root with a special marker \checkmark . The initial tree T_0 is defined as expected, from (u_1, v_1) . The PCP instance has a solution iff we can reach a data tree with root label \checkmark . Notice that all guards and the invariant are empty.

For termination we can modify the above proof in order to ensure that executions that do not correspond to partial solutions, are infinite. More precisely, if U, V as above is a partial solution, but Uu_i, Vv_i is not, then we use a DTP rule associated with (u_i, v_i) that forces an infinite execution. In this way, termination will hold iff the PCP instance has a solution. \square

We end this section with a remark on the decidability of termination from an initial set of trees. First we notice that – already without data – DTPRS can simulate so-called *reset Petri nets* [15]. These are Petri nets (or equivalently, multi-counter automata without zero test) with additional transitions that can reset places (equivalently, counters) to zero. They can be represented by trees of depth 2, where nodes at depth one represent places, and their respective number of children (leaves) is the number of tokens on that place. A DTPRS (without data) can easily simulate increments, decrements and resets (using deletion in DTPRS). It is known that so-called *structural termination* for reset Petri nets is undecidable [18], i.e., the question whether there are infinite computations from *any* initial configuration, is undecidable. This implies:

PROPOSITION 4. *The following question is undecidable: Given a DTPRS (\mathcal{R}, Δ) , is there some tree T_0 satisfying Δ and an infinite computation $T_0 \rightarrow T_1 \rightarrow \dots$ in (\mathcal{R}, Δ) ? This holds already for non-recursive DTD in Δ and without data constraints in DTPs.*

5. POSITIVE DTPRS

In this section we consider *positive DTPRS*, a fragment of DTPRS for which we show that termination and pattern reachability are decidable.

From Proposition 2, we know that in order to get decidability, the DTD in the static invariant Δ must be non-recursive. For a non-recursive DTD, there is some B such that every tree satisfying the DTD has depth bounded by B . In the following, we assume the existence of such a bound B . Also from Proposition 2, we know that for decidability we need to consider only positive guards and positive data invariants.

However, from Theorem 3, we know that these restrictions alone do not suffice to achieve decidability. We need to disallow long linear orders created with the help of data. For this, we introduce a last restriction, called *simple-path bounded*, which is defined in the following.

Let $T = (V, E, \text{root}, \ell)$ be a data tree. The graph $G(T)$ associated with T is the undirected graph obtained by adding to V the set of data values occurring in T , and adding to E the links between a leaf labeled by a data value and the node representing that value (see also Figure 3). Formally, $G(T) = (V', E')$, where $V' = V \cup \{\ell(v) \mid \ell(v) \in \mathcal{D}\}$ and $E' = E \cup \{\{v, d\} \mid \ell(v) = d \in \mathcal{D}\}$. A *simple path* of T is a simple path in $G(T)$, i.e. a sequence of vertices v_1, \dots, v_n in $G(T)$ such that for all $i \neq j$, $\{v_i, v_{i+1}\} \in E'$ and $v_i \neq v_j$. The length of a path v_1, \dots, v_n is $n - 1$.

Formally, a DTPRS (\mathcal{R}, Δ) is a *positive DTPRS* with set of initial trees Init , if:

- **non-recursive-DTD:** the DTD in the static invariant Δ is non-recursive. In particular, trees satisfying the DTD have depth bounded by some $B > 0$.

- **positive:** all guards in \mathcal{R} and the data invariant in Δ are positive Boolean combinations of DTPs. The DTD in Δ is positive as well.
- **simple-path bounded:** there exists $K > 0$ such that the length of any simple path in any $T \in \mathcal{T}_{\mathcal{R}}^*(T_0)$ for any $T_0 \in \text{Init}$, is bounded by K .

Notice that the third condition above implies that all data trees have depth bounded by K . So we always assume that $B \leq K$. Notice also that in positive DTPRS, the data value inequality is *allowed* in DTPs, that is, we can state that two data values are different.

The *Play.com* example in Section 3 satisfies the first 2 conditions above. However, in general, the third condition is not satisfied. PIDs can create a long path: a cart can be linked to a product, linked to another cart linked to another products etc. So the number of carts or the number of products needs to be bounded (unless a cart can contain at most one product). On the other hand, Name and Total are fresh data values, they cannot be used as links. At last, CID can be used in different carts and orders, but as a cart or order is associated to a unique customer, it cannot create long links. More formally, if the system can handle only C active carts at a time (but the number of orders is unlimited), then the system has simple paths bounded by $12C + 7$. If there are at most D different products in the catalog, then the system has simple paths bounded by $12D + 7$. Finally, if each customer can have only one active cart at a time (but she can have many orders), and each cart has at most one product, then the system has simple paths bounded by 14. Any of these restrictions can be described using only positive rules. The rest of the section is devoted to the proof of the following result:

THEOREM 5. *Given a positive DTPRS (\mathcal{R}, Δ) , the pattern reachability and the termination problem are decidable.*

We prove Theorem 5 by using the framework of *well-structured transition systems (WSTS)* [1, 13], which has been applied to DTPRS *without* data in [15]. We recall briefly some definitions. A WSTS is a triple $(S, \rightarrow, \preceq)$ such that S is an (infinite) state space, \preceq is a *well-quasi-ordering*³ (wqo for short) on S , and \rightarrow is the transition relation on S . It is required that \rightarrow is *compatible w.r.t.* \preceq : for any $s, t, s' \in S$ with $s \rightarrow t$ and $s \preceq s'$, there exists $t' \in S$ such that $s' \rightarrow t'$ and $t \preceq t'$.

Let $\mathcal{T}_{B,K}$ denote the set of data trees whose depths are bounded by B and lengths of simple paths are bounded by K . From the definition of positive DTPRS, we know that $\mathcal{T}_{\mathcal{R}}^*(\text{Init}) \subseteq \mathcal{T}_{B,K}$.

In the following, we show Theorem 5 by showing that $(\mathcal{T}_{B,K}, \rightarrow, \preceq)$ is a WSTS.

We define the binary relation \preceq on $\mathcal{T}_{B,K}$. Let $T_1 = (V_1, E_1, \text{root}_1, \ell_1), T_2 = (V_2, E_2, \text{root}_2, \ell_2) \in \mathcal{T}_{B,K}$, then $T_1 \preceq T_2$ if there is an injective mapping ϕ from V_1 to V_2 such that

- root preservation: $\phi(\text{root}_1) = \text{root}_2$,
- parent-child relation preservation: $(v_1, v_2) \in E_1$ iff $(\phi(v_1), \phi(v_2)) \in E_2$,
- tag preservation: If $\ell_1(v) \in \Sigma$, then $\ell_1(v) = \ell_2(\phi(v))$,

³A wqo \preceq is a reflexive, transitive and well-founded relation with no infinite antichain.

- data value (in)equality preservation: If $v_1, v_2 \in V_1$ and $\ell_1(v_1), \ell_1(v_2) \in \mathcal{D}$, then $\ell_2(\phi(v_1)), \ell_2(\phi(v_2)) \in \mathcal{D}$, and $\ell_1(v_1) = \ell_1(v_2)$ iff $\ell_2(\phi(v_1)) = \ell_2(\phi(v_2))$.

It is easy to see that \preceq is reflexive and transitive, so it is a quasi-order. In the following, we first assume that \preceq is a wqo on $\mathcal{T}_{B,K}$ and show that \longrightarrow is compatible with \preceq , in order to prove Theorem 5. We show in Section 5.3 that \preceq is indeed a wqo: for any infinite sequence of data trees $T_0, T_1, \dots \in \mathcal{T}_{B,K}$, there are $i < j$ such that $T_i \preceq T_j$.

5.1 Well-structure of positive DTPRS

Let (\mathcal{R}, Δ) be a positive DTPRS.

PROPOSITION 6. *Let $T_1, T'_1, T_2 \in \mathcal{T}_{B,K}$, $T_1 \xrightarrow{R} T_2$ for some $R \in \mathcal{R}$, and $T_1 \preceq T'_1$. Then there exists $T'_2 \in \mathcal{T}_{B,K}$ such that $T'_1 \xrightarrow{R} T'_2$ and $T_2 \preceq T'_2$.*

PROOF. Let $R = (L, G, \mathcal{Q}, \mathcal{F}, \chi)$. Taking an injective mapping $\phi : T_1 \rightarrow T'_1$ preserving the root, parent-child relation, tag, and data (in)equality relation, and an injective matching $\psi : L \rightarrow T_1$ satisfying the data constraint *cond* of L , we have an injective matching $\phi \circ \psi : L \rightarrow T'_1$ which respects the parent-child, tags and data (in)equality relation. Hence *cond* is satisfied by $\phi \circ \psi$ too. As G is positive, if G is true at T_1 wrt. ϕ , then it is true at T'_1 wrt. $\phi \circ \psi$ as well. Applying the rule R to T'_1 wrt. $\phi \circ \psi$, we get a tree T'_2 such that $T_2 \preceq T'_2$. As both the DTD and the data invariant in Δ are positive and T_2 fulfills Δ , so does T'_2 . Thus $T'_1 \xrightarrow{R} T'_2$. \square

Consequently, we have shown that \longrightarrow is compatible wrt. \preceq in $\mathcal{T}_{B,K}$, thus $(\mathcal{T}_{B,K}, \longrightarrow, \preceq)$ is a WSTS.

In the following, we prove that $(\mathcal{T}_{B,K}, \longrightarrow, \preceq)$ satisfies some additional computability conditions, needed to show the decidability of pattern reachability and termination.

First consider pattern reachability. To get the decidability of this problem, from Theorem 3.6 in [13], we need to show that $(\mathcal{T}_{B,K}, \longrightarrow, \preceq)$ has effective pred-basis. A WSTS $(S, \longrightarrow, \preceq)$ has *effective pred-basis* if there exists an algorithm that computes for any state $s \in S$ the finite basis $pb(s)$ of the upward closed set $\uparrow \text{Pred}(\uparrow s)$. Here, $\uparrow I = \{s' \in S \mid \exists s \in I \text{ s.t. } s \preceq s'\}$ denotes the upward closure of I wrt. \preceq , and $\text{Pred}(I) = \{s \in S \mid \exists t \in I, s \longrightarrow t\}$ the set of immediate predecessors of states in I . A basis of an upward-closed set I is a minimal set I^b such that $I = \bigcup_{x \in I^b} \uparrow x$. Recall that whenever \preceq is a wqo, the basis I^b of an upward closed set I is finite.

PROPOSITION 7. *$(\mathcal{T}_{B,K}, \longrightarrow, \preceq)$ has effective pred-basis.*

A solution for reachability of a given DTP P from an initial set of data trees Init is obtained by backward exploration: we start with I^0 as the set of data trees matching P and satisfying Δ . Then compute iteratively the upward closed sets $I^{n+1} = I^n \cup (\text{Pred}(I^n) \cap \Delta)$ by representing each set through its finite basis. Since the sequence I^n is increasing by construction, and since \preceq is a wqo, the sequence must be finite and termination can be effectively tested. If $I^n = I^{n+1}$ then it suffices to check whether $I^n \cap \text{Init} = \emptyset$. Notice that we did not impose any restriction on the set Init of the initial trees. We need to test the existence of a data tree from $\mathcal{T}_{B,K}$ satisfying an (arbitrary) Boolean combination on DTPs and an (arbitrary) DTD. This problem is in

general undecidable [8], but becomes decidable in the special case where trees are of bounded depth [8]. Here we need to talk in addition about trees from $\mathcal{T}_{B,K}$, but we can apply the same proof ideas as in [8] in order to infer decidability.

Now consider the termination problem. From Theorem 4.6 in [13], to show the decidability of termination problem from a single initial tree T_0 , it is sufficient to show that $(\mathcal{T}_{\mathcal{R}}^*(T_0), \xrightarrow{R}, \preceq)$ has effective *Succ*, i.e. for each $T \in \mathcal{T}_{\mathcal{R}}^*(T_0)$, the set $\text{Succ}(T) := \{T' \mid T \longrightarrow T'\}$ is computable. Then we can compute the *finite reachability tree* starting with T_0 : we compute trees T s.t. $T_0 \xrightarrow{*} T$ and we stop whenever we find $T \preceq T'$ along some branch.

It is not hard to see that $\text{Succ}(T)$ contains only a finite number of equivalence classes induced by the quasi-order \preceq . Since the DTPRS (\mathcal{R}, Δ) is not able to distinguish between two distinct data trees belonging to the same equivalence class, by selecting one data tree from each equivalence class, we can get a finite representation of $\text{Succ}(T)$, therefore, $(\mathcal{T}_{\mathcal{R}}^*(T_0), \longrightarrow, \preceq)$ has effective *Succ*.

5.2 Tree Decompositions

In order to prove that \preceq is a wqo over $\mathcal{T}_{B,K}$, we first represent a data tree T as a (labeled) undirected graph $G_\ell(T)$, then we encode $G_\ell(T)$ into a tree (without data) of *bounded depth* using the concept of tree decompositions. Define a binary relation \leq on (labeled) trees of bounded depth as follows: $T_1 \leq T_2$ if there is an injective mapping from T_1 to T_2 preserving the root, the tags, and the parent-child relation. It is known that \leq is a wqo on labeled trees of bounded depth *without data* [15].

Let \mathcal{G}_K be the set of labeled graphs with the lengths of all simple paths bounded by K . We show that \preceq on $\mathcal{T}_{B,K}$ corresponds to the induced subgraph relation (formally defined later) on \mathcal{G}_K , and the fact that \leq is a wqo for labeled trees of bounded depth implies that the induced subgraph relation is a wqo on \mathcal{G}_K .

Given a data tree $T = (V, E, \text{root}, \ell) \in \mathcal{T}_{B,K}$, the *labeled undirected graph representation* $G_\ell(T)$ of T is obtained from $G(T)$, the graph associated to T , by adding labels encoding information of data tree nodes (tag, depth ...). Formally, $G_\ell(T)$, is a $((\Sigma \cup \{\#\}) \times [B+1]) \cup \{\$\}$ -labeled (where $[B+1] = \{0, 1, \dots, B\}$) undirected graph (V', E', ℓ') defined as follows,

- $V' = V \cup \{\ell(v) \mid v \in V, \ell(v) \in \mathcal{D}\}$,
- $E' = E \cup \{\{v, d\} \mid v \in V, \ell(v) = d \in \mathcal{D}\}$,
- If $\ell(v) \in \Sigma$, then $\ell'(v) = (\ell(v), i)$, otherwise, $\ell'(v) = (\#, i)$, where i is the depth of v in T . In addition, $\ell'(d) = \$$ for each $d \in V' \cap \mathcal{D}$.

Let Σ_G denote $((\Sigma \cup \{\#\}) \times [B+1]) \cup \{\$\}$.

For Σ_G -labeled graphs, we define the induced subgraph relation as follows. Let $G_1 = (V_1, E_1, \ell_1), G_2 = (V_2, E_2, \ell_2)$ be two Σ_G -labeled graphs, then G_1 is an *induced subgraph* of G_2 (denoted $G_1 \sqsubseteq G_2$) iff there is an injective mapping ϕ from V_1 to V_2 such that

- label preservation: $\ell_1(v_1) = \ell_2(\phi(v_1))$ for any $v_1 \in V_1$,
- edge preservation: let $v_1, v'_1 \in V_1$, then $\{v_1, v'_1\} \in E_1$ iff $\{\phi(v_1), \phi(v'_1)\} \in E_2$.

From the definition of the labeled graph representation of data trees, it is not hard to show that the induced subgraph relation \sqsubseteq corresponds to the relation \preceq on data trees.

PROPOSITION 8. Let $T_1, T_2 \in \mathcal{T}_{B,K}$, then $T_1 \preceq T_2$ iff $G_\ell(T_1) \subseteq G_\ell(T_2)$.

Now we show how to encode any Σ_G -labeled graph belonging to \mathcal{G}_K into a labeled tree of bounded depth by using tree decompositions.

Let $G = (V, E, \ell)$ be a connected Σ_G -labeled graph, then a *tree decomposition* of G is a quadruple $T = (U, F, r, \theta)$ such that:

- (U, F, r) is a tree with the tree domain U , the parent-child relation F , and the root $r \in U$,
- $\theta : U \rightarrow 2^V$ is a labeling function attaching each node $u \in U$ a set of vertices of G ,
- For each edge $\{v, w\} \in E$, there is a node $u \in U$ such that $\{v, w\} \subseteq \theta(u)$,
- For each vertex $v \in V$, the set of nodes $u \in U$ such that $v \in \theta(u)$ constitutes a connected subgraph of T .

The sets $\theta(v)$ are called the *bags* of the tree decomposition.

The *depth* of a tree decomposition $T = (U, F, r, \theta)$ is the depth of the tree (U, F, r) and the *width* of T is defined as $\max\{|\theta(u)| - 1 \mid u \in U\}$. The *tree-width* of a graph $G = (V, E)$ is the minimum width of tree decompositions of G . For a tree decomposition of width K of a graph G , without loss of generality, we assume that each bag is given by a sequence of vertices of length $K + 1$, $v_0 \dots v_K$, with possible repetitions, i.e. possibly $v_i = v_j$ for some $i, j : i \neq j$ (tree decompositions in this form are sometimes called ordered tree decompositions).

THEOREM 9 ([19, 6]). If $G \in \mathcal{G}_K$, then G has a tree decomposition with both depth and width bounded by K .

PROOF. Let $G = (V, E, \ell) \in \mathcal{G}_K$ and $T = (V, E_T, r)$ be a depth-first-search tree of G with $r \in V$ as the root. Then T is of depth at most K . For each $v \in V$, let $\theta(v)$ be the union of $\{v\}$ and the set of all ancestors of v in T , then (V, E_T, r, θ) is a tree decomposition of G of depth at most K and width at most K . \square

As a matter of fact, the converse of Theorem 9 holds as well.

PROPOSITION 10. If G has a tree decomposition of width $\leq A$ and depth $\leq B$, then the length of any simple path of G is bounded by $(A + 2)^B + \sum_{1 \leq i \leq B} (A + 2)^i$.

So generally speaking, for a class of graphs, all simple paths are length-bounded for each graph in the class iff there is a tree decomposition of bounded depth and width for each graph in the class.

Now we describe how to encode labeled graphs by labeled trees using tree decompositions.

Let $G = (V, E, \ell) \in \mathcal{G}_K$ be a Σ_G -labeled graph, and $T = (U, F, r, \theta)$ be a tree decomposition of G with width K and depth at most K . Remember that each $\theta(u)$ is represented as a sequence of exactly $K + 1$ vertices, and $[K + 1] = \{0, \dots, K\}$. Define

$$\Sigma_{G,K} := (\Sigma_G)^{K+1} \times 2^{[K+1]^2} \times 2^{[K+1]^2} \times 2^{[K+1]^2}.$$

We transform $T = (U, F, r, \theta)$ into a $\Sigma_{G,K}$ -labeled tree $T' = (U, F, r, \eta)$, which encodes in a uniform way the information about G (including edge relations and vertex labels). $\eta : U \rightarrow \Sigma_{G,K}$ is defined as follows. Let $\theta(u) = v_0 \dots v_K$, then $\eta(u) = (\ell(v_0) \dots \ell(v_K), \bar{\lambda})$, where $\bar{\lambda} = (\lambda_1, \lambda_2, \lambda_3)$,

- $\lambda_1 = \{(i, j) \mid 0 \leq i, j \leq K, v_i = v_j\}$,
- $\lambda_2 = \{(i, j) \mid 0 \leq i, j \leq K, \{v_i, v_j\} \in E\}$,
- If $u = r$, then $\lambda_3 = \emptyset$, otherwise let u' be the parent of u in T and $\theta(u') = v'_0 \dots v'_K$, then $\lambda_3 = \{(i, j) \mid 0 \leq i, j \leq K, v'_i = v'_j\}$.

5.3 Well-quasi-ordering for data trees

The encoding of labeled graphs into labeled trees establishes a connection between the wqo \leq of labeled trees and the induced subgraph relation (\sqsubseteq) of labeled graphs.

PROPOSITION 11. Let G_1, G_2 be two Σ_G -labeled graphs with tree-width bounded by K , and T_1, T_2 be two tree decompositions of width K of resp. G_1, G_2 , then the two $\Sigma_{G,K}$ -labeled trees T'_1, T'_2 obtained from T_1, T_2 satisfy that: If $T'_1 \leq T'_2$, then $G_1 \sqsubseteq G_2$.

PROOF. Let $G_i = (V_i, E_i, \ell_i)$, $T_i = (U_i, F_i, r_i, \theta_i)$ and $T'_i = (U_i, F_i, r_i, \eta_i)$ ($i = 1, 2$). Suppose that $T'_1 \leq T'_2$. Then there is an injective mapping ϕ from U_1 to U_2 preserving the root, the parent-child relation and the node-labels.

Define an injective mapping $\pi : V_1 \rightarrow V_2$ as follows:

For $v \in V_1$, select some $u \in U_1$ such that $\theta_1(u) = v_0 \dots v_K$ and $v = v_i$ for some i . Writing $\theta_2(\phi(u)) = v'_0 \dots v'_K$, we let $\pi(v) = v'_i$.

First we show that π does not depend upon the choice of i such that $v = v_i$, neither on the choice of $u \in U_1$ such that $v \in \theta_1(u)$. The former holds because $\eta_1(u) = \eta_2(\phi(u))$ (and in particular the component λ_1 is preserved), hence if $v_i = v_j$, then we also have $v'_i = v'_j$.

For the latter, notice that the λ_3 component of $\eta_1(u) = \eta_2(\phi(u))$ is preserved, hence the choice of u or of its father is irrelevant. Now, the set $\{u \in U_1 \mid v \in \theta_1(u)\}$ is a connected subgraph of T_1 by definition of tree decomposition, hence π does not depend upon the choice of $u \in U_1$.

Now we show that π is injective. Let v_2 be a vertex of G_2 . Because of the preservation of λ_1 , no two different vertices v, v' of G_1 with $v, v' \in \theta(u)$ can satisfy $\pi(v) = \pi(v') = v_2$. Because of the preservation of λ_3 , no two different vertices v, v' with $v \in \theta(u)$ and $v' \in \theta(u')$ with u father of u' can satisfy $\pi(v) = \pi(v') = v_2$. Again, as the set $\{u \in U_1 \mid v_2 \in \theta(u)\}$ is a connected subgraph of T_2 , it means that π is injective.

We finish the proof by showing that π preserves the node-labels and edge relations.

Node-label preservation: Suppose $\pi(v) = v'$. Then there exists some $u \in U_1$ such that $\theta_1(u) = v_0 \dots v_K$, $v = v_i$ for some i , $\theta_2(\phi(u)) = v'_0 \dots v'_K$, and $v' = v'_i$. Since $\eta_1(u) = \eta_2(\phi(u))$, $\ell_1(v_0) \dots \ell_1(v_K) = \ell_2(v'_0) \dots \ell_2(v'_K)$, it follows that $\ell_1(v) = \ell_1(v_i) = \ell_2(v'_i) = \ell_2(v')$.

Edge relation preservation: We show that $\{v, w\} \in E_1$ iff $\{\pi(v), \pi(w)\} \in E_2$ for any $v, w \in V_1$.

If $\{v, w\} \in E_1$, there exists $u \in U_1$ such that $\theta_1(u) = v_0 \dots v_K$, $v = v_i$ and $w = v_j$ for some i, j . So $(i, j) \in \lambda_2(u)$ in T'_1 . Then $(i, j) \in \lambda_2(\phi(u))$. Let $\theta_2(\phi(u)) = v'_0 \dots v'_K$, then $\{v'_i, v'_j\} \in E_2$. Consequently $\{\pi(v), \pi(w)\} = \{v'_i, v'_j\} \in E_2$.

If $\{\pi(v), \pi(w)\} \in E_2$, then there exists $u' \in U_2$ such that $\pi(v), \pi(w) \in \theta_2(u')$. Without loss of generality, we can choose u' at minimal depth such that $\pi(v), \pi(w) \in \theta_2(u')$. It means that for instance, the father u'' of u' satisfies $\pi(v) \notin \theta_2(u'')$. Since $U'_2 = \{u''' \in U_2 \mid \pi(v) \in \theta_2(u''')\}$ is connected,

it means that U'_2 is entirely contained in the subtree rooted at u' . By contradiction, if there does not exist $u \in U_1$ such that $\phi(u) = u'$, then $\phi(U_1) \cap U'_2 = \emptyset$. On the other hand, according to the definition of π , there is $u \in U_1$ such that $v \in \theta_1(u)$ and $\pi(v) \in \theta_2(\phi(u))$. So $\phi(u) \in \phi(U_1) \cap U'_2$, a contradiction. Thus there is $u \in U_1$ such that $u' = \phi(u)$. Let $\theta_1(u) = v_0 \dots v_k$ and $\theta_2(u') = v'_0 \dots v'_k$, by injectivity of π , we have $\pi(v) = v'_i$, $\pi(w) = v'_j$, $v = v_i$, $w = w_j$ for some i, j . Then $(i, j) \in \lambda_2(u') = \lambda_2(u)$, which proves that $\{v, w\}$ is an edge of G_1 . \square

Now we are ready to show that \preceq is a wqo for $\mathcal{T}_{B,K}$.

Let T_0, T_1, \dots be an infinite sequence of data trees from $\mathcal{T}_{B,K}$. Consider the infinite sequence of $\Sigma_{G,K}$ -labeled trees T'_0, T'_1, \dots obtained from the tree decompositions (with width K and depth at most K) of graphs $G_\ell(T_0), G_\ell(T_1), \dots$. Then there are $i, j : i < j$ such that $T'_i \leq T'_j$, because \leq is a wqo for labeled trees of depth at most K . So $G(T_i) \sqsubseteq G(T_j)$ from Proposition 11, and $T_i \preceq T_j$ from Proposition 8. We thus prove following theorem.

THEOREM 12. \preceq is a well-quasi-ordering over $\mathcal{T}_{B,K}$.

6. VERIFICATION OF TEMPORAL PROPERTIES

Until now we considered only two properties for static analysis: termination and pattern reachability. (Non-)reachability of a DTP can be expressed easily in Tree-LTL [5], which corresponds roughly to linear time temporal logics where atomic propositions are DTPs⁴. We show in this section that allowing for runs of unbounded length makes the validation of (even simple) Tree-LTL properties undecidable, even without data:

PROPOSITION 13. *It is undecidable whether a TPRS with initial (data-free) tree T_0 satisfies a given Tree-LTL formula.*

The fact that Tree-LTL is undecidable does not disallow us to verify quite complicated properties. We show on the *Play.com* example how to proceed: it suffices to encode in the system the property we want to check with additional tags and check for pattern reachability. For example, suppose that we want to verify whether a customer can add some product after the bill was processed. For that, we add new tags $\#, 1, 2, 3$ to the alphabet, and we add one child to *Play.com* labeled by 1 in the initial tree. We add one rule which checks that the additional tag is 1 and selects one cart in the payment mode. The outcome of the rule is to change the tag from 1 to 2, and to append $\#$ as child of the selected cart. We add another rule which checks that the tag is 2. The outcome of the rule is to change the tag from 2 to 3, and to append a new product with one item in the inventory below PCatalog with a special marker $\#$ as brother of PId. Now one can reach in the new system a tree with a cart marked $\#$ and with a product with PId X such that there exist a Product with PId X in the PCatalog which is marked by $\#$ iff a customer can add some product after the bill was generated in the original system. The former property is a pattern reachability problem, which we proved to be decidable.

⁴Such formulas use actually free variables in patterns, which are then quantified universally. This is consistent with the approach of testing whether a model satisfies the negation of a formula.

7. BOUNDED MODEL-CHECKING DTPRS AND RECURSION-FREE GAXML

Recall that [5] shows that the largest decidable fragment of GAXML that can be model-checked w.r.t. Tree-LTL properties is the recursion-free one. Absence of recursion in GAXML roughly means (1) disallowing recursive DTDs (as we do here) and (2) imposing that no function is called more than once, on any execution path. On the other hand, one can use negated DTPs in this fragment.

In this section we consider *bounded model checking* for DTPRS: Given a DTPRS (\mathcal{R}, Δ) , a set of initial trees *Init*, a DTP P and a bound N (encoded in unary) we ask whether there is some T_0 satisfying *Init* and some T s.t. P matches T and $T_0 \xrightarrow{\leq N} T$. We show the following result:

THEOREM 14. *Bounded model-checking for DTPRS is NexpTime-complete.*

Theorem 14 can be actually extended to bounded model-checking Tree-LTL properties. Bounded model-checking of a Tree-LTL formula φ with a bound N is the problem checking whether a counter-example for φ holds in $\leq N$ -steps. For instance, bounded model-checking for $G \neg P$ with a bound N is to check whether the DTP P can be reached in $\leq N$ steps.

For the upper bound we show how to encode a DTPRS (\mathcal{R}, Δ) with the given bound N into a recursion-free GAXML system, and use the upper bound provided by [5]. We recall that [5] provides a *simply* exponential bound in the number of transition steps of the recursion-free GAXML system. Also notice that the DTPRS in Theorem 14 are not supposed to be positive – the lower bound relies on negations of DTPs.

The basic idea of the reduction from bounded model-checking DTPRS to recursion-free GAXML is the following: we “guess” on-the fly the rules R_1, R_2, \dots, R_M , $M \leq N$, that are applied on a successful path of the DTPRS, and use function labels for pinpointing the nodes used by the matching of the corresponding locators. Suppose that nodes of locators have identifiers. A node having a child (leaf) labeled by the function call $!(i, R_i, w)$ with $i \leq M$, $R_i \in \mathcal{R}$ and w an identifier within the locator L_i of rule R_i , is “guessed” to correspond to node w in the matching of the locator L_i when applying rule R_i . Notice that node can have several function calls $(i, R_i, v), (j, R_j, w)$ attached to it (but then, $i \neq j$, since the matching of L_i should be injective).

We use the DTD in the invariant Δ in order to ensure that (1) each of the (polynomially many) function labels $!(i, R_i, w)$ occurs at most once at any time point, and (2) we use GAXML call/return guards for ensuring that calls related to rule R_i are only performed after all function calls (j, R_j, v) with $j < i$, have been completed, for each i . Checking (2) is done by forbidding the presence of function calls $!(j, R_j, v)$ and $?(j, R_j, v)$ with $j < i$, whenever (i, R_i, w) is called. Similarly, when the result of a call $?(i, R_i, w)$ is returned, we forbid that it contains some label $!(j, R_j, v)$ or $?(j, R_j, v)$ with $j < i$.

Applying rule R_i means calling all functions $!(i, R_i, w)$ one-by-one (say in DFS order) and performing the associated actions. When we call the first function with index i , its guard also checks that the locator L_i was properly guessed. A rename action must be simulated, since GAXML has no renaming facility: a call $!(i, R_i, w)$ with w labeled by ren_b

in L_i has as effect to attach label b_i as a child (leaf) of node w . Checking which is the current label of a node is done by using negative guards: we look for a child b_i that has no sibling c_k with $k > i$. A delete action must be simulated, too, since GAXML has no deletion. A call $!(i, R_i, w)$ with w labeled by del in L_i has as effect to return a node with tag del . This might be syntactically inconsistent if the current node is a data node. However, standard encoding tricks can remedy this problem. For simplicity, let us assume that the tag del is always appended as a sibling of the node that is supposed to be deleted in DTPRS.

Finally, the append action is done as in GAXML, by performing the query and attaching the result. Here, we need to take care about the nodes that added via a forest $F \in \mathcal{F}$, resp. a query. For such nodes, we must “guess” some attached function calls. In both cases we use external GAXML functions: for example, we can simulate the addition of an annotated copy of F via an external call, and use Δ for checking that the right F was added. For query answers we may split a query Q in polynomially many copies, where for some of the copies, the *head* has attached external function calls. Their role is to generate (sets) of functions of type $!(i, R, v)$ attached to nodes in *head*.

The last point is that we need to adapt the GAXML upper bound in order to take care of nodes marked by del : this can be done e.g. by extending the notion of matching tree patterns in such a way that none of the nodes to which tree patterns are matched, nor their ancestors, are allowed to have a child labeled by the tag del . It can be easily checked that the complexity checking a Tree-LTL formula for recursion-free GAXML still holds with this extended notion of matching. The reason is that the proof is based on small models obtained by taking tree prefixes of the bigger model. Obviously, the absence of children with tag del is preserved by taking tree prefixes.

The lower bound is adapted from the 2-NexpTime lower bound proof for recursion-free GAXML. We only recall the rough idea here.

The main ingredient of the proof is to create/check lists of length 2^n . This is done using data values, similarly as in the proof of Theorem 3. A “list” of length k corresponds to a tree of depth 2, where each node at depth one has 2 children, with distinct data values d_i, d_{i+1} . If each data value d_i occurs twice (except for d_1 and d_{k+1} , which occur only once) we get a linear order, i.e. a list. Using n queries we can compute n steps of transitive closure and thus verify that $k = 2^n$. Obviously, this suffices for encoding a $(2^n \times 2^n)$ tableau representing a computation of a 2^n -time bounded TM. Details are fairly easy to complete.

8. CONCLUSION

In this paper, we defined a rich class of systems describing active documents, possibly with recursive calls. We show that this class of systems is easy to use and powerful, demonstrating it on the MailOrder example. We studied the boundary of decidability for different properties and restrictions of the active documents. Namely, we show that termination from one document and pattern reachability from a set of documents are both decidable for *positive* DTPRSs, which are DTPRSs where the DTD is non-recursive, there is no negative guard or data invariants, and *simple paths* are bounded. We showed that without these restrictions, the problem is undecidable. We also show that the problem is

undecidable for more complex properties (Tree LTL or termination from a set of active documents). Nevertheless, we also demonstrate on the MailOrder example that one can find bugs with our method.

Compared with GAXML [5], the respective restrictions used to get decidability (positiveness and non-recursion) are incomparable. We showed however a reduction from (not necessarily positive) rewriting-length bounded DTPRS to recursion-free GAXML.

Considering further work it seems possible to get decidability results for another (incomparable) class of systems, namely DTPRSs whose new data variables do not get mutually distinct fresh values, but possibly arbitrary data values.

9. REFERENCES

- [1] P. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS'96*, pages 313–321. IEEE, 1996.
- [2] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezzine. Monotonic abstraction for programs with dynamic memory heaps. In *CAV'08*, volume 5123 of *LNCS*, pages 341–354. Springer, 2008.
- [3] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *PODS'04*, pages 35–45. ACM, 2004.
- [4] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.
- [5] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. In *PODS '08*, pages 221–230, New York, NY, USA, 2008. ACM.
- [6] A. Blumensath and B. Courcelle. On the monadic second-order transduction hierarchy. HAL Archive, 2009. <http://hal.archives-ouvertes.fr/hal-00287223/fr>.
- [7] P. Bouyer, N. Markey, J. Ouaknine, P. Schnoebelen, and J. Worrell. On termination for faulty channel machines. In *STACS'08*, volume 1 of *Leibniz International Proceedings in Informatics*, pages 121–132, 2008.
- [8] C. David. Complexity of data tree patterns over XML documents. In *MFC'S '08*, pages 278–289, Berlin, Heidelberg, 2008. Springer-Verlag.
- [9] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT'09*, pages 252–267. ACM, 2009.
- [10] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.
- [11] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In *PODS'06*, pages 90–99. ACM, 2006.
- [12] A. Deutsch and V. Vianu. WAVE: Automatic verification of data-driven web services. *IEEE Data Eng. Bull.*, 31(3):35–39, 2008.
- [13] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
- [14] X. Fu, T. Bultan, and J. Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.
- [15] B. Genest, A. Muscholl, O. Serre, and M. Zeitoun. Tree pattern rewriting systems. In *ATVA'08*, pages

- 332–346. Springer, 2008.
- [16] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences.*, pages 1152–1163. Springer-Verlag, 2008.
 - [17] R. Hull, M. Benedikt, V. Christophides, and S. Jianwen. E-services: a look behind the curtain. In *PODS'03*, pages 1–14. ACM, 2003.
 - [18] R. Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci.*, 297(1-3):337–354, 2003.
 - [19] J. Nešetřil and P. O. de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *Eur. J. Comb.*, 27(6):1022–1041, 2006.
 - [20] J. Wang and A. Kumar. A framework for document-driven workflow systems. In *Business Process Management*, pages 285–301, 2005.

APPENDIX

Proof of Proposition 2:

In both cases we encode a 2-counter machine with counters a, b . In the first one, a configuration $(q, n_a, n_b) \in Q \times \mathbb{N} \times \mathbb{N}$ is encoded by a tree with root labeled q and two subtrees, one of the form $a^{n_a}a_s$, and the other of the form $b^{n_b}b_s$. E.g. a zero test on the first counter corresponds to checking that the root has a child labeled a_s . Decrementing the first counter in state q (and going to state q') is done using the locator $[q](-[a]([a_s]))$, where the additional labels are: $ren_{q'}$ for the root, ren_{a_s} for the a -node and del for the a_s -node.

With non-recursive DTDs we can encode a configuration (q, n_a, n_b) by a tree of depth one, with root labeled by q , and n_a (n_b , resp.) leaves labeled by a (b , resp.). The zero test is now done using a negative guard (e.g. "no a -leaf") or a negative invariant. In the latter case we split a transition in 2 steps: first we relabel the root by a transition from that state; second, we perform the corresponding rewriting as before. The invariant states that whenever the root is labeled by a transition corresponding to a zero test of counter c , the tree has no c -leaf ($c \in \{a, b\}$). \square

Proof of Proposition 7:

It is sufficient to consider $\min(Pred(\uparrow T))$, the set of minimal elements wrt. \preceq in $Pred(\uparrow T)$, for each tree $T \in \mathcal{T}_{B,K}$.

Fix a rule $R = (L, G, \mathcal{Q}, \mathcal{F}, \chi)$ with

$$L = (V_L, E_L, root_L, \ell_L, \tau_L, cond_L, \ell'_L)$$

such that ℓ'_L attaches additional labels $\{append, ren_a, del\}$, $\mathcal{Q} = \{Q_1, \dots, Q_m\}$ ($Q_i = body_i \rightsquigarrow head_i$), and $\mathcal{F} = \{F_1, \dots, F_n\}$.

Let $T_1 = (V_1, E_1, root_1, \ell_1) \in \min(Pred(\uparrow T))$, then

$$\exists T' \text{ such that } T_1 \xrightarrow{R} T' \text{ wrt. some } \phi : L \rightarrow T_1 \text{ and } T \preceq T' \text{ via some } \psi : T \rightarrow T'. \quad (*)$$

In the following, we show that the size of T_1 (number of nodes) is bounded by the following constant (we actually show even more, by exhibiting T_1 satisfying Δ):

$$B^2 ((|\Sigma| + 1) \max(\Delta))^B (|L| + |G| + |T| \max_i |Q_i|).$$

Thus a finite basis, which is a finite subset of $\min(Pred(\uparrow T))$, is computable.

Let $T' = (V', E', root', \ell')$. Then V' consists of four disjoint subsets,

- $V'_1 = \{\phi(v) \mid v \in V_L, v \text{ not labeled by } del\}$,
- $V'_2 = node^{-1}(V'_1)$, where $node^{-1}(V'_1)$ is the set of nodes $w \in V_1 \setminus \phi(V_L)$ such that the lowest ancestor of w in $\phi(V_L)$ is in V'_1 .
- V'_3 contains distinct copies of F_j , excluding the leaves labeled by those Q_i ,
- V'_4 contains distinct copies of the nodes of the forest $Q_i(T_1)$, one for each node labeled by Q_i in each copy of F_j .

The node set of T_1 consists of $V'_1, V'_2, V_3 = \{\phi(v) \mid v \in V_L, v \text{ labeled by } del\}$, and $V_4 = node^{-1}(V_3)$.

Now we consider an upper bound on the size of T_1 that are sufficient to allow T_1 satisfying $(*)$,

- To guarantee the matching ϕ from L to T_1 :
The nodes in $V'_1 \cup V_3 = \phi(V_L)$ and all their ancestors in T_1 are sufficient.
Note that in L , ancestor relations \parallel may occur, so the

inclusion of the ancestors of nodes in $V'_1 \cup V_3 = \phi(V_L)$ is necessary.

Size: $B|\phi(V_L)| = B|L|$;

- To witness that G is satisfied over T_1 wrt. ϕ :
 G is a positive Boolean combination of DTPs. To witness the satisfaction of each DTP P_i in G , we need keep a matching ϕ_i from P_i to T_1 and all the ancestors of nodes of $\phi_i(P_i)$ in T_1 .

Size: $B|G|$;

- To guarantee that $T \preceq T'$:
 - Keep $(V'_1 \cup V'_2) \cap \psi(V_T)$ and all their ancestors in T_1 ,
 - At most $|T|$ instantiations of $head_i$ on T_1 by matchings from $body_i$ to T_1 wrt. ϕ are sufficient. The ancestors of all the nodes of T_1 in these instantiations should be preserved as well.

Size: $B|T| + B|T||body_i| \leq B|T| \max_i |Q_i|$.

- Finally, to satisfy the DTD in Δ , T_1 should be completed into a data tree of size at most (c.f. [5])

$$B \cdot (|\Sigma| + 1) \max(\Delta))^B |T_1|,$$

where $\max(\Delta)$ is the maximum integer used in the definition of DTD in Δ .

Thus a sufficient upper bound for the size of T_1 is

$$B^2 ((|\Sigma| + 1) \max(\Delta))^B (|L| + |G| + |T| \max_i |Q_i|).$$

\square

Proof of Proposition 10.

Let $G = (V, E)$ and $T = (W, F, r, \theta)$ be a tree decomposition of G of width at most A and depth at most B .

Let $P = v_1 \dots v_n$ be a path in G , and $w_1 \dots w_n$ be a trace of P in T such that $v_i \in \theta(w_i)$, $w_i = w_{i+1}$ or there is a path in T from w_i to w_{i+1} such that for each $w \neq w_{i+1}$ on the path, $v_i \in \theta(w)$.

Because all bags are of size at most $A + 1$, each bag can only occur at most $A + 1$ times on the sequence $w_1 \dots w_n$.

Let B_0 be the minimal depth of w_i 's. Then there is only one bag at depth B_0 , say w , occurring on the sequence $w_1 \dots w_n$.

Let w_{i_1}, \dots, w_{i_l} ($l \leq A + 1$, $i_j < i_{j+1}$) be all the occurrences of w on the sequence $w_1 \dots w_n$. Then all the bags on each sub sequence $w_{i_j+1} w_{i_j+2} \dots w_{i_{j+1}-1}$ is at depth no less than $B_0 + 1$. By induction hypothesis, each subsequence $w_{i_j+1} w_{i_j+2} \dots w_{i_{j+1}-1}$ is of length at most

$$(A + 2)^{B-B_0-1} + \sum_{1 \leq i \leq B-B_0-1} (A + 2)^i,$$

thus

$$\begin{aligned} n &\leq l + (l + 1) \left((A + 2)^{B-B_0-1} + \sum_{1 \leq i \leq B-B_0-1} (A + 2)^i \right) \\ &\leq (A + 2) \left(1 + (A + 2)^{B-1} + \sum_{1 \leq i \leq B-1} (A + 2)^i \right) \\ &= (A + 2)^B + \sum_{1 \leq i \leq B} (A + 2)^i. \end{aligned}$$

\square

Proof of Proposition 13:

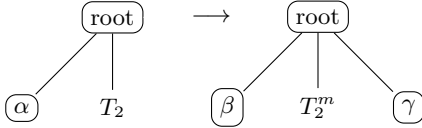
We give a reduction from the *exact* reachability problem, i.e. checking whether a tree T_2 can be reached *exactly* from a tree T_1 via a TPRS \mathcal{R} . This problem was shown to be undecidable in [15] by a reduction from the reachability problem on reset Petri nets⁵.

We reduce the exact reachability problem for \mathcal{R} to checking the Tree-LTL formula $\varphi = \mathbf{G}(P_1 \rightarrow \mathbf{F} P_2)$ (P_1 and P_2 are DTPs) for a TPRS \mathcal{R}' and initial tree T_0 .

Let Σ be the set of tags of \mathcal{R} , and let Σ_m be a disjoint copy of Σ . We will use five new tags $root, \alpha, \beta, \gamma, \delta \notin \mathcal{T}$, thus $\mathcal{T}' = \mathcal{T} \cup \Sigma_m \cup \{root, \alpha, \beta, \gamma, \delta\}$.

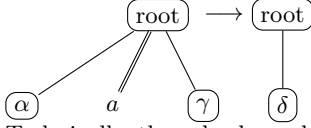
The starting tree T_0 consists of a root with one child labeled α , and one other child tree that equals T_1 .

The TPRS \mathcal{R}' consists of \mathcal{R} (adapted in order to take the additional root into account), plus two kinds of new rules (with empty guards):



In the rule above, T_2^m means that we use the tag copy Σ_m for T_2 . Technically, the rule renames α by β , renames each tag $a \in \Sigma$ in T_2 by $a_m \in \Sigma_m$, and appends a new node with tag γ .

The second type of additional rules is the following (here, $a \in \Sigma$ parametrizes the rule):



Technically, the rule above deletes node α and the subtree rooted at a , and rename γ by δ .

Finally, the Tree-LTL property to be checked on (\mathcal{R}', Δ') and initial tree T_0 is $\varphi = \mathbf{G}(P_1 \rightarrow \mathbf{F} P_2)$, where $P_1 = [\text{root}](\gamma)$ and $P_2 = [\text{root}](\delta)$. It is easy to see that $T_1 \xrightarrow{*} T_2$ via \mathcal{R} iff \mathcal{R}' does not satisfy φ from the initial tree T_0 : TP P_1 can be generated only if from T_1 we can reach a tree via \mathcal{R} , that contains T_2 as prefix. But then, TP P_2 cannot be generated only if from T_1 we reach *exactly* T_2 (cf. second rule). \square

⁵Although the TPRS model used in [15] is slightly more general than the present model, it is easily seen that in the reduction in [15] we use the TPRS model presented here.