Static Analysis of Active XML Systems

Serge Abiteboul  
INRIA-Futurs, U. Paris Sud  
Serge.Abiteboul@inria.fr

Luc Segoufin  
INRIA-Futurs, U. Paris Sud  
http://www-roq.inria.fr/~segufin

Victor Vianu  
U.C. San Diego  
vianu@cs.ucsd.edu

Abstract

Active XML is a high-level specification language tailored to data-intensive, distributed, dynamic Web services. Active XML is based on XML documents with embedded function calls. The state of a document evolves depending on the result of internal function calls (local computations) or external ones (interactions with users or other services). Function calls return documents that may be active, so may activate new subtasks. The focus of the paper is on the verification of temporal properties of runs of Active XML systems, specified in a tree-pattern based temporal logic, Tree-LTL, that allows expressing a rich class of semantic properties of the application. The main results establish the boundary of decidability and the complexity of automatic verification of Tree-LTL properties.

1 Introduction

Data-intensive, distributed, dynamic applications are pervasive on today’s Web. The reliability of such applications is often critical, but their logical complexity makes them vulnerable to potentially costly bugs. Classical automatic verification techniques operate on finite-state abstractions that ignore the critical semantics associated with data in such applications. The need to take into account data semantics has spurred interest in studying static analysis tasks in which data is explicitly present (see related work). In this paper, we make a contribution in this direction by investigating automatic verification in a model tightly integrating the XML and Web service paradigms. Specifically, we consider Active XML, a high-level specification language tailored to data-intensive Web applications, and Tree-LTL, a tree-based temporal logic that can express a rich class of temporal properties of such applications. We establish the boundary of decidability and the complexity of automatic verification in this setting. In particular, we isolate an important fragment of Active XML (sufficient to describe a large class of applications) for which the verification of temporal properties is decidable.

Active XML documents [2, 4] (AXML for short) are XML documents [23] with embedded function calls interpreted as Web service calls [24]. In the spirit of [18, 21], a document is seen as a process that evolves in time. A function call is seen as a request to carry out a subtask whose result may lead to a change of state in the document. Our goal is to analyze the behavior of such systems, which is especially challenging because the presence of data induces infinitely many states.

To illustrate the kind of applications we target, consider a mail order processing system. The arrival of a new order corresponds to the initiation of a new task. At each moment, the system is running a possibly large number of orders, initiated by different users. Processing each order may involve various subtasks. For instance, a credit check may be requested from a credit service, and its outcome determines how the order proceeds. In our approach, the entire mail order system, as well as each individual order, are seen as AXML documents that evolve in time.

Our goal is to analyze the behavior of AXML systems, and in particular to verify temporal properties of their runs. For instance, one may want to verify whether some static property (e.g., all ordered products are available) and some dynamic property (e.g., an order is never delivered before payment is received) always hold. The language Tree-LTL allows to express a rich class of such properties.

A main contribution of the paper is to carefully design an appropriate abstraction of AXML that is expressive enough to describe meaningful applications, and can also serve as a convenient formal vehicle for studying decidability and complexity boundaries for verification in the model. This has lead to Guard AXML, that we briefly describe next.

In Guard AXML (GAXML for short), document
trees are unordered. With ordered trees, verification quickly becomes intractable. GAXML distinguishes between internal and external services. An internal service is a service that is completely specified, i.e., its precise semantics is known. External services capture interactions with other services and with users.

For these, only partial information on their input and output type is available. Finally, the most novel feature of the model in the AXML context is a guard mechanism for controlling the initiation and completion of subtasks (formally function calls). Guards are Boolean combinations of tree patterns. They facilitate specifying applications driven by complex workflows and, more generally, they provide a very useful programming paradigm for active documents.

An AXML system consists of AXML documents running on different peers and interacting between them and with the external world. To simplify the presentation, we consider here single-peer systems. We will mention how the model can be extended to multipeer systems and our results applied to this larger setting that actually motivated this work.

Our main results establish the boundary of decidability of satisfaction of Tree-LTL properties by GAXML systems. We obtain decidability by disallowing recursion in GAXML systems, which leads to a bound on the number of total function calls in runs. We prove that for such recursion-free GAXML, the satisfaction of Tree-LTL formulas is Co-2EXPTIME-complete. We also consider various relaxations of the non-recursiveness restriction and show that they each lead to undecidability. This establishes a fairly tight boundary of decidability of verification. At the same time, we show that certain limited but useful verification tasks remain decidable even with recursion. For instance, we provide a decidable sufficient condition for safety with respect to a Boolean combination of tree patterns. We also show that it is decidable whether a state satisfying a Boolean combination of tree patterns can be reached within a specified number of steps in a run.

Related work Most of previous works on static analysis on XML (with data values) was dealing with documents that do not evolve with time. Typically one would consider the consistency problem for XML specifications using DTDs and (foreign) key constraints [6, 7], the query containment problem [3] or the type checking problem [8]. This motivated studies of automata and logics on strings and trees over infinite alphabets [20, 12, 9]. See [22] for a survey on related issues.

Previous works also considered the evolution of documents. For instance, static analysis was considered in [1] for a restricted monotone AXML language, positive AXML. Their setting is very different from ours as their systems are monotone. In contrast, we consider a broader verification task for nonmonotone systems.

Verification of temporal properties of Web services has mostly been considered using models abstracting away data values (see [17] for a survey). Verification of data-intensive Web services was studied in [13, 15], and a verifier implemented [14]. As in our case, this work takes into account data and establishes the boundary of decidability and complexity of verification for a restricted class of services and properties expressed in a temporal logic. While this is related in spirit to the present work, the technical differences stemming from the AXML setting render the two investigations orthogonal.

Organization After presenting in Section 2 the GAXML model and the language Tree-LTL, we present in Section 3 the decidability and complexity results for recursion-free GAXML services. Relaxations of non-recursiveness are considered in Section 4, and shown to lead to undecidability. The decidability results on safety and bounded reachability are also presented in Section 4. The paper concludes with a brief discussion. The proofs, and a detailed running example, are provided in an appendix.

2 The GAXML model

We formalize in this section the GAXML model. To simplify the presentation, we consider a system with a single peer (we discuss this issue in Section 5). To illustrate our definitions, we use fragments of a Mail Order GAXML processing system, more fully described in the appendix.

In this paper, trees are unranked and unordered. A forest is a set of trees. The notions of node, child, descendant, ancestor, and parent relations between nodes are defined in the usual way. A subtree of a tree \( T \) is the tree induced by \( T \) on the set of all descendants of a particular node.

We assume given the following disjoint infinite sets: nodes \( N \) (denoted \( n, m \)), tags \( \Sigma \) (denoted \( a, b, c, \ldots \)), function names \( F \), data values \( D \) (denoted \( \alpha, \beta, \ldots \)) data variables \( V \) (denoted \( X, Y, Z, \ldots \)), possibly with subscripts. The set \( F \) is the union of two disjoint sets of marked function symbols \( F^i \) and \( F^s \), where \( F^i \) is a set of symbols of the form \( !f \), and \( F^s = \{ ?f | !f \in \}

\( \mathcal{F} \)). Intuitively, \( /f \) labels a node where a call to function \( f \) can be made (possible call), and \( ?f \) labels a node where a call to \( f \) has been made and some result is expected (running call).

A Guard AXML (GAXML) document is a tree whose internal nodes are labeled with tags in \( \Sigma \) and whose leaves are labeled by either tags, function names, or data values. A GAXML forest is a set of GAXML trees. An example\(^1\) of GAXML document is given in Figure 1.

To avoid repetitions of isomorphic sibling subtrees, we define the notion of reduced tree. Two trees \( T_1 \) and \( T_2 \) are isomorphic if there exists a bijection from the nodes of \( T_1 \) to the nodes of \( T_2 \) that preserves the edge relation and the labeling of nodes. A tree is reduced if it contains no isomorphic sibling subtrees. Clearly, each tree \( T \) can be reduced by eliminating duplicate isomorphic sibling subtrees, and the result is unique up to isomorphism. We henceforth assume that all trees considered are reduced, unless stated otherwise. However, forests may generally contain multiple isomorphic trees.

**Patterns** We use patterns as the building blocks for guards for controlling the activation of function calls and as a basis for our query language. A pattern is a forest of tree patterns. A tree pattern is a tree whose edges and nodes are labeled. An edge label indicates a child (\( / \)) or descendant (\( // \)) relationship. A node label either restricts the label of the node or is a variable denoting a data value. A constraint consisting of a Boolean combination of (in)equalities between the variables and/or data constants may also be given. In particular, we can specify joins (equality of data values). A tree pattern is evaluated over a tree by mapping all the nodes of a pattern to the tree while respecting the constraints of its tree patterns. In particular, the root of the tree pattern is mapped to the root of the tree. The mapping need not be injective: two different nodes of the pattern may be mapped to the same node in the tree. The definition of the evaluation of patterns over forests extends the above in the natural way. Since the notion of pattern is rather standard, its formal definition is relegated to the appendix. An example is given in Figure 2 (a). The pattern shown there expresses the fact that the value \( \text{Order-Id} \) is not a key. It does not hold on the GAXML document of Figure 1. (Indeed, we want \( \text{Order-Id} \) to be a key).

In some guards and queries, we use patterns that are evaluated relative to a specified node in the tree. More precisely, a relative pattern is a pair \( (P, \text{self}) \) where \( P \) is a pattern and \( \text{self} \) is a node of \( P \). A relative pattern \( (P', \text{self}) \) is evaluated on a pair \( (F, n) \) where \( F \) is a forest and \( n \) is a node of \( F \). Such a pattern forces the node \( \text{self} \) in the pattern to be matched to \( n \). Figure 2 (b) provides an example of relative pattern. The pattern shown there checks that a product that has been ordered occurs in the catalog. It holds in the GAXML document of Figure 1 when evaluated at the unique node labeled \( \text{Bill} \).

We also consider Boolean combinations of (relative) patterns. The (relative) patterns are matched independently of each other and the Boolean operators have their standard meaning. If a variable \( X \) occurs in two different patterns \( P \) and \( P' \) of the Boolean combination then it is treated as quantified existentially for \( P \) and independently quantified for \( P' \).

![Figure 2: Two patterns](image-url)

**Queries** As previously mentioned, patterns are also used in queries, as shown next. A query is defined by pairs of patterns, a *Body* and a *Head*. When evaluated on a forest, the matchings of *Body* define a set of valuations of the variables. The *Head* pattern then specifies how to construct the result from these valuations. A particular node ("constructor" node below) specifies a form of nesting.

More formally, a query is an expression \( \text{Body} \rightarrow \text{Head} \) where *Body* and *Head* are patterns such that for each \( H \) in *Head*,

- its internal nodes have labels in \( \Sigma \) and its leaves have labels in \( \Sigma \cup \mathcal{F} \cup \mathcal{V} \);
- there is no repeated variable in \( H \) and each variable occurring in it also occurs in *Body* and
- there is one designated node \( c \) in \( H \) called the *constructor* node, such that the subtree rooted
at $c$ contains all variables in $H$. In graphical representations, this constructor node is marked with set parenthesis. (In absence of variables in $H$, the constructor may be omitted.)

As for patterns, we consider queries evaluated relative to a specified node in the input tree. A relative query is defined like a query, except that its body is a relative pattern ($P$, self). An example of relative query is given in Figure 3. The label of the constructor node is Process-bill.

![Figure 3: Example of a relative query](image)

Let $F$ be a forest and $Q = Body \to H$ a query with a single tree for head. Let $M$ be the set of matchings of Body into $F$. Let $c$ be the constructor node of $H$ and $H_c$ the subtree of $H$ rooted at $c$. For each matching $\mu \in M$, let $\mu(H_c)$ be an isomorphic copy of $H_c$ with new nodes, in which every variable label $X$ occurring in $H$ is first replaced by $\mu(X)$ and the tree is next reduced. Then the result $Q(F)$ is the forest obtained by replacing $c$ in $H$ by the reduced forest $\{\mu(H_c) \mid \mu \in M\}$. Note that if $M = \emptyset$ then $c$ is simply removed. Observe also that, when $c$ is not the root, $Q(F)$ is a single-tree forest. When $c$ is the root, the forest may have 0, 1 or more trees. Now consider a query $Q = Body \to H_1, \ldots, H_n$. Then $Q(F) = \cup Q_i(F)$ where for each $i$, $Q_i = Body \to H_i$.

A relative query is evaluated on a pair $(F, n)$ where $F$ is a forest and $n$ is a node of $F$. The result $Q(F, n)$ is defined as for queries, except that matchings of the body must map self to $n$.

**Remark 2.1** The constructor node provides explicit control over nesting of results. Note that this can be

seen as syntactic sugaring in AXML, since the same effect can be achieved using function calls. However, the explicit constructor node is convenient from a specification viewpoint. Observe also that one could consider nesting of constructor nodes, in the spirit of group-by operators. Such an extension, which for simplicity we do not consider here, would not affect our results.

Consider the evaluation of the query of Figure 3 on the GXML document of Figure 1 at the unique node labeled $\text{Bill}$. There is a unique matching of the Body pattern and the result is the Head pattern of the query with $X$ replaced by $\text{Nikon}$ and $Y$ by 199 (and no more parenthesis for Process-bill).

**DTD** Trees used by a GXML system may be constrained using DTDs and tree pattern formulas. For DTDs, we use a typing mechanism that restricts, for each tag $a \in \Sigma$, the labels of children that $a$-nodes may have. As our trees are unordered we use Boolean combinations of statements of the form $|b| \geq k$ for $b \in \Sigma \cup \mathcal{F} \cup \{\text{dom}\}$ and $k$ a non-negative integer. Validity of trees and of forests relative to a DTD is defined in the standard way. Details are omitted.

**Schema and instance** A GXML schema $S$ is a tuple $(\Phi_{int}, \Phi_{ext}, \Delta)$ where

- The set $\Phi_{int}$ contains a finite set of internal function specifications.
- The set $\Phi_{ext}$ contains a finite set of external function specifications.
- $\Delta$ provides static constraints on instances of the schema. It consists of a DTD and a Boolean combination of patterns.

We next detail $\Phi_{int}$ and $\Phi_{ext}$. For each $f \in \mathcal{F}$, let $a_f$ be a new distinct label in $\Sigma$. Intuitively,
$a_f$ will be the root of a subtree where a call to $f$ is being evaluated. (This subtree may be seen as some work space for the evaluation of the function.) Each function $f$ of $\Phi_{\text{int}}$ is specified as a tuple $\langle \text{arg}(f), \text{kind}(f), \gamma(f), \rho(f), \text{ret}(f) \rangle$ where:

- $\text{arg}(f)$ (the input query) is a (relative) query. Intuitively, its role is to define the argument of a call to $f$, which is also the initial state in the evaluation of $f$. If the query defining the argument is relative, self binds to the node at which the call $!f$ is made.

- $\text{kind}(f) \in \{\text{non-continuous, continuous}\}$. If $f$ is non-continuous, a call to $f$ is deleted once the answer is returned. If $f$ is continuous, the call is kept after the answer is returned, so $f$ can be called again.

- $\gamma(f)$ (the call guard) is a Boolean combination of relative patterns. A call to $f$ can only be made if $\gamma(f)$ holds. (Observe that negative conditions are allowed.)

- $\rho(f)$ (the return guard) is a Boolean combination of patterns rooted at $a_f$. The result of a call to $f$ can only be returned when the return guard is satisfied.

- $\text{ret}(f)$ (the return query) is a query rooted at $a_f$.

**Example 2.2** We continue with our running example. The function $\text{Bill}$ used in Figure 1 is specified as follows. It is internal and non-continuous. Its call guard is the pattern in Figure 2 (b). The argument query is the query in Figure 3. Assuming that $\text{Invoice}$ is an external function eventually returning $\text{Payment}$ (with product and amount paid) the return guard and query of $\text{Bill}$ are:

![Diagram of return guard and query]

Return guard | Return query

Each function $f$ in $\Phi_{\text{ext}}$ is specified similarly, except that the return guard $\rho(f)$ and the return query $\text{ret}(f)$ are missing. Intuitively, an external call can return any answer at any time. Its answer can only be constrained by $\Delta$.

An instance $I$ over a GAXML schema $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ is a pair $(T, \text{eval})$, where $T$ is a GAXML forest and $\text{eval}$ an injective function over the set of nodes in $T$ labeled with $?f$ for some $f \in \Phi_{\text{int}}$ such that:

1. For each $n$ with label $?f$, $\text{eval}(n)$ is a tree in $T$ with root label $a_f$.
2. Every tree in $T$ with root label $a_f$ is $\text{eval}(n)$ for some $n$ labeled $?f$.

An instance of $S$ is valid if it satisfies $\Delta$.

**Runs** Let $I = (T, \text{eval})$ and $I' = (T', \text{eval}')$ be instances of a GAXML schema $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$. The instance $I'$ is a possible next instance of $I$, denoted $I \vdash I'$, iff $I'$ is obtained from $I$ in one of the following ways:

- **External call** there exists some node $n$ in $T \in T$, labeled $!f$ for $f \in \Phi_{\text{ext}}$, such that $\gamma(f)(T, n)$ holds, where $\gamma(f)$ is the call guard of $f$; and $I'$ is obtained from $I$ by changing the label of $n$ to $?f$.

- **Internal call** This is like for external function except that $f \in \Phi_{\text{int}}$. Furthermore, we add to the graph of $\text{eval}$ the pair $(n, T')$ where $T'$ is a tree consisting of a root $a_f$ connected to the forest that is the result of evaluating the argument query $\text{arg}(f)$ on input $(T, n)$. (All nodes occurring in $T'$ are new.)

- **Return of internal call** There is some node $n$ labeled $?f$ in some tree of $T$, where $f \in \Phi_{\text{int}}$, such that $T = \text{eval}(n)$ contains no running call labels $?g$ and the return guard of $f$ is true on $T$. Then $I'$ is obtained from $I$ as follows:

  - evaluate the return query $\text{ret}(f)$ on $T$ and add the resulting forest as a sibling of the node $n$;
  - remove $\text{eval}(n)$ from $T$ and $n$ from the domain of $\text{eval}$
  - if $f$ is non-continuous remove the node $n$, otherwise change $n$’s label to $!f$.

- **Return of external call** There exists some node $n$ labeled $?f$ in some tree of $T$, for $f \in \Phi_{\text{ext}}$. Then $I'$ is obtained as for the return of internal calls, except that (i) there is no corresponding running computation to remove from $\text{eval}$ and (ii) the result (a forest with labels in $\Sigma \cup \mathcal{F} \cup \mathcal{D}$ appended as a sibling to $n$) is chosen arbitrarily (but note that if $I'$ is required to be valid, then $\Delta$ also constrains the result of the external call).

Figure 4 shows a possible next instance for the instance of Figure 1 after an internal call has been made.
to !Bill. Recall the specification of Bill from Example 2.2. The call was enabled as the guard of !Bill is true on the instance of Figure 1 (see Figure 2). As !Bill is an internal call, the subtree a\textsubscript{Bill} contains the result of the query defining !Bill (see Figure 3). The dotted arrow indicates the function \texttt{eval}.

We will typically be interested in verifying temporal properties of runs of a given system. An initial instance of \(S\) is an instance of \(S\) consisting of a single tree whose root is not a function call and for which there is no running call. An instance \(I\) is blocking if there is no instance \(I'\) such that \(I \vdash I'\). A run of \(S\) is an infinite sequence \(I_0, I_1, \ldots, I_i, \ldots\) of instances over \(S\) such that \(I_0\) is an initial instance of \(S\) and for each \(i \geq 0\), either \(I_i \vdash I_{i+1}\) or \(I_i\) is blocking and \(I_{i+1} = I_i\). Note that, for uniformity, we force all runs to be infinite by repeating a blocking instance forever if it is reached. A run is valid if all of its instances satisfy \(\Delta\). For a run \(\rho\), we denote by \text{adom}(\rho) the set of data values occurring in \(\rho\), which may be infinite due to external function calls.

**Temporal properties** As mentioned in the introduction, we are interested in verifying certain properties of runs of a GAXML systems. These may include generic desirable properties, such as always reaching a successful final instance (blocking with no active function calls), as well as properties specific to the particular application, such as “no product is delivered before it is paid in the right amount”.

To express such temporal properties of runs, we use patterns connected by Boolean and temporal operators. This yields the language Tree-LTL (and branching-time variants Tree-CTL or Tree-CTL\(^*\)). More precisely, we use the auxiliary notion of \text{QPattern} (for quantified pattern). A \text{QPattern} is an expression \(P(\tilde{X})\) where \(P\) is a pattern and \(\tilde{X}\) some of its variables, designated as \text{free}. All other variables will be seen as quantified existentially, locally to \(P\). (So logically, \(P(\tilde{X})\) may be seen as \(\exists Y(P)\), where \(Y\) is the set of variables occurring in \(P\) and not \(\tilde{X}\).) The syntax of Tree-LTL formulas is defined by the following grammar:

\[\varphi ::= \text{QPattern} | \varphi \land \varphi | \varphi \lor \varphi | \neg \varphi | \varphi \mathrel{U} \varphi | X\varphi\]

where \(U\) stands for \text{until} and \(X\) for \text{next}, with the usual semantics, e.g., see [16]. Given a Tree-LTL formula \(\varphi\), its free variables are the free variables of its patterns. A Tree-LTL sentence is an expression \(\forall \tilde{X}, \varphi(\tilde{X})\), where \(\varphi\) is a Tree-LTL formula and \(\tilde{X}\) are the free variables of \(\varphi\). (As previously mentioned, variables that are not free are existentially quantified locally to each pattern.)

Whenever convenient, we use as shorthand additional temporal operators expressible using \(X\) and \(U\), such as \(F\) (eventually) and \(G\) (always).

We now turn to the semantics of Tree-LTL. Intuitively, a sentence \(\forall \tilde{X}, \varphi(\tilde{X})\) holds for a schema \(S\) if \(\varphi(\tilde{X})\) holds on every valid run of \(S\) with every interpretation of \(\tilde{X}\) into the active domain of the run. More formally, consider first the case when \(\varphi\) has no free variables. Consider a run \(\rho\) of \(S\). Satisfaction of a pattern without free variables by an instance was defined previously. Therefore, patterns can be treated as propositions and we can use the standard semantics of LTL to define when \(\rho\) satisfies \(\varphi\), denoted by \(\rho \models \varphi\). Consider now a Tree-LTL sentence \(\sigma = \forall \tilde{X}, \varphi(\tilde{X})\). For a run \(\rho\) of \(S\), we say that \(\rho\) satisfies \(\forall \tilde{X}, \varphi(\tilde{X})\), and denote this by \(\rho \models \forall \tilde{X}, \varphi(\tilde{X})\), if \(\rho\) satisfies \(\varphi(h(\tilde{X}))\) for each valuation \(h\) of \(\tilde{X}\) into \text{adom}(\rho).

We say that \(S\) satisfies \(\sigma\), denoted \(S \models \sigma\), if every valid run of \(S\) satisfies \(\sigma\).

Two examples of Tree-LTL formulas are given below.

Every mail order is eventually completed (delivered or rejected):

\[\forall X [G(\text{Main} \rightarrow F(\text{MailOrder} \lor \text{Main}))] \]

Every product for which a correct amount has been paid is eventually delivered (note that the variable \(Z\) is implicitly existentially quantified in the left pattern):

\[\forall X \forall Y [G(\text{Main} \rightarrow F(\text{MailOrder}))] \]

![Figure 5: Some Tree-LTL formulas.](image-url)
3 Recursion-free GAXML

Most of our positive results are obtained under the assumption that AXML services are recursion-free. This restriction essentially bounds the number of function calls in a run of the system. The external functions clearly are a source of difficulty for enforcing non-recursiveness syntactically, since an external function $f$ may return some data with a call to some external function $g$, and $g$ some data with a call to $f$. To circumvent this, we must assume some signature information on external functions. We do this by including in the specification of each external function $f$ the set $\text{fun}(f)$ of functions that are allowed to appear in the results of calls to $f$. The definition of valid run is modified so that this restriction is obeyed. For internal functions $f$ and $g$, $g$ is in $\text{fun}(f)$ if $!g$ occurs in the result of the argument or return query of $f$. (This can be checked syntactically by inspecting the head of the respective queries.)

To define non-recursiveness, we use the auxiliary notion of call graph that captures (syntactic) dependencies between function calls in the schema. Let $S = (\Phi_{int}, \Phi_{ext}, \Delta)$ be a GAXML schema. The call graph $G$ of $S$ is a directed graph whose nodes are $\Phi_{int} \cup \Phi_{ext}$ and there is an edge from $f$ to $g$ if $g \in \text{fun}(f)$.

**Definition 3.1** Let $S = (\Phi_{int}, \Phi_{ext}, \Delta)$ be a GAXML schema. We say that $S$ is recursion-free iff the following hold:

(i) the DTD of $\Delta$ is non-recursive,

(ii) no function call $!f$ occurs more than once in a tree satisfying the DTD of $\Delta$, 

(iii) no function of $S$ is continuous, and

(iv) the call graph of $S$ is acyclic.

As mentioned above, the definition of recursion-free schema is meant to enforce a static bound on the number of function calls made in a valid run.

While conditions (i), (iii) and (iv) achieve this by prohibiting the immediate causes of recursion, condition (ii) deals with another source of unbounded calls, the presence of an arbitrary number of them in the initial instance or in answers to external function calls. Condition (ii) could be relaxed without loss by allowing a bounded number of calls to each function rather than a single one. Also note that condition (ii) restricts each tree in an instance, but not the instance as a whole. Thus, a function call may appear in several different trees of the same instance.

The main result of the section is that satisfaction of a Tree-LTL sentence by a recursion-free GAXML schema is co-2EXPTIME-complete. For each recursion-free GAXML schema, we first show that every valid run reaches a blocking instance after at most an exponential number of calls. We will also see that it is only necessary to consider runs over "small" instances. Finally, we will present a nondeterministic algorithm for checking a temporal property on a run. Altogether, we obtain a co-2EXPTIME bound that can be shown to be tight.

The first result shows that if $S$ is recursion-free, then each valid run of $S$ reaches a blocking instance after a number of transitions that is exponential in the size of the schema. This is a consequence of the fact that without recursion, only finitely many calls to each function can be made. The details can be found in the appendix.

**Proposition 3.2** Let $S = (\Phi_{int}, \Phi_{ext}, \Delta)$ be a recursion-free GAXML schema. There exists a non-negative integer $k$, exponential in $|\Phi_{int} \cup \Phi_{ext}|$, such that all valid runs of $S$ reach a blocking instance in at most $k$ transitions.

Let $S$ be a recursion-free GAXML schema. A pre-run of $S$ is a finite prefix of a run ending in the first occurrence of its blocking instance. We say that a pre-run of $S$ satisfies a Tree-LTL sentence $\varphi$ iff its infinite extension satisfies $\varphi$. We note the following useful fact. Its proof uses standard Büchi automata techniques, after replacing each pattern in $\varphi$ by a suitable proposition (see Appendix).
Proposition 3.3 Given a pre-run $\rho = I_0, \ldots, I_k$ of $S$ and a Tree-LTL sentence $\varphi$, one can check whether $\rho$ satisfies $\varphi$ using a non-deterministic algorithm in time $O(|\rho|^{|\varphi|})$.

The next proposition is the key to our decision algorithm. It shows that only runs with small instances need to be considered. This is the most difficult part of the proof and is achieved by carefully identifying a "small" set of nodes sufficient to witness satisfaction of the patterns needed for the run to be valid and satisfy $\varphi$ (see Appendix for proof).

Proposition 3.4 If there exists a valid pre-run of $S$ satisfying $\varphi$, then there exists a valid pre-run of the same length satisfying $\varphi$, such that each of its instances has size doubly exponential in $|\varphi|$ and $S$.

The previous results immediately yield a decision procedure. A recursion-free GXML schema does not satisfy $\varphi$ iff there is valid pre-run violating $\varphi$. This pre-run can be guessed nondeterministically (its size is at most doubly exponential by Proposition 3.4 and Proposition 3.2) and then $\neg \varphi$ can be checked using Proposition 3.3. The matching lowerbound is difficult and technical and is stated in the following proposition. Its proof is given in the appendix. Intuitively, we consider a non-deterministic Turing Machine $M$ running in time $2^{2^n}$ on inputs of size $n$. We exhibit a Tree-LTL sentence $\varphi$ and a recursion-free GXML schema $S$ such that $\varphi$ holds on all valid runs of $S$ iff $M$ rejects $w$. The difficulty of the proof lies in simulating a Turing machine running in time $2^{2^n}$ with a recursion-free GXML schema using only $n$ functions and hence with runs of length $2^n$. This requires a very efficient use of the available functions (see Appendix).

Proposition 3.5 It is $\text{co-2-NEXPTIME}$-hard to check whether a recursion-free GXML schema satisfies a Tree-LTL sentence.

We now have the main result of the section:

Theorem 3.6 It is $\text{co-NEXPTIME}$-complete to decide, given a recursion-free GXML schema $S$ and a Tree-LTL sentence $\varphi$, whether each valid run of $S$ satisfies $\varphi$.

Remark 3.7 While the worst-case $\text{co-NEXPTIME}$ complexity of verification we have just shown may appear daunting, the complexity is likely to be much lower in many practical situations. For example, for GXML schemas whose call graph is a tree (a likely occurrence when functions model a hierarchical set of tasks) the complexity goes down to $\text{co-NEXPTIME}$. Within the broader landscape of static analysis, this is quite reasonable. For instance, recall that even satisfiability of Barnays-Schönfinkel FO sentences, a much simpler question, already has complexity $\text{NEXPTIME}$ [10].

Using similar techniques, we can show decidability of the following useful static analysis tasks for recursion-free GXML (see Appendix for proofs).

Corollary 3.8 The following are decidable in co-$\text{2NEXPTIME}$ for a recursion-free GXML schema $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$:

- Successful termination: each valid run of $S$ ends in a blocking instance with no running function calls.
- Typechecking: for every run of $S$, if the initial instance satisfies $\Delta$, then every instance in the run satisfies $\Delta$.

Remark 3.9 The above notion of typechecking is quite strict, since it declares a violation even if it is caused by the result of a call to an external function (in other words, a service will typecheck only if at any point in the run, any result of an external function call is acceptable wrt $\Delta$). A more lenient variant would typecheck subject to the assumption that results from calls to external functions do not cause violations. The decidability result of Corollary 3.8 can be easily extended to this variant.

4 Beyond recursion-free

In this section we prove that decidability of satisfaction of a Tree-LTL formula by a GXML schema is lost even under minor relaxations of non- recursiveness. However, certain restricted but useful verification tasks remain decidable. We provide several such results in the second part of this section.

Undecidability We next consider relaxations of each of the recursion-free conditions and show that each such relaxation induces undecidability of satisfaction of Tree-LTL sentences. Specifically, we consider each of the following extensions: allowing (1) recursive DTDs, (2) an unbounded number of function calls in trees satisfying $\Delta$, (3) continuous functions, (4) a cyclic call graph.
For (1), undecidability is a simple consequence of the fact that satisfiability of Boolean combination of patterns in the presence of a DTD is already undecidable [11]. The first result concerns extensions (2-3). We prove a strong undecidability result, showing that even reachability of an instance satisfying a single positive pattern without variables becomes undecidable with any of these extensions. Furthermore, the result holds for schemas without data constraints and using no external functions. The proof is by reduction from the implication problem for functional and inclusion dependencies, known to be undecidable. See the appendix for details.

**Theorem 4.1** It is undecidable, given a positive pattern $P$ without variables and a GXML schema $S$ with no data constraints or external functions, satisfying the non-recursiveness conditions relaxed by any of (2) or (3) above, whether some instance satisfying $P$ is reachable in a valid run of $S$.

In order to show that Condition (4) also yields undecidability, we use the fact that, with cyclic call graphs, we can generate arbitrarily long sequences of running function calls allowing us to code two-counter automata (see Appendix). Note that this result holds even without any data values.

**Theorem 4.2** It is undecidable, given a positive pattern $P$ without variables and a GXML schema $S$ with no data values and no external functions, satisfying the non-recursiveness conditions relaxed by allowing a cyclic call graph, whether some instance satisfying $P$ is reachable in a valid run of $S$.

**Remark 4.3** The results for extensions (3) and (4) point to significant qualitative differences between recursion obtained by using continuous functions, and by allowing cyclic call graphs. Theorem 4.2 suggests that the latter is much more powerful. The distinction is further highlighted by considering the instance-dependent variant of verification: given a GXML schema $S$, an initial instance $I$ of $S$, and a Tree-LTL formula $\varphi$, does every run starting from $I$ satisfy $\varphi$? An immediate consequence of the proof of Theorem 4.2 is that this is undecidable for GXML schema with cyclic call graphs (even with no data values and only internal functions). On the other hand, it is easily seen that this is decidable for arbitrary GAXML schemas with continuous internal functions (but cyclic call graph). This follows from the fact that the fixed initial instance renders the state space finite, which is not the case if cyclic call graphs are allowed.

The above results show that relaxations of the non-recursiveness requirements quickly lead to strong forms of undecidability. Orthogonally, one might wonder if decidability can be preserved for recursion-free schemas for more powerful queries or temporal properties. We next show that this is not the case.

We first consider an extension to the patterns used so far in the GAXML model, allowing negative sub-patterns. Specifically, let us allow labeling by $\neg$ one subtree of the pattern, with the safety restriction that all variables occurring in the negative subtree must also occur positively in the pattern. The semantics is the natural one: a match requires the positive part of the subtree to be matched to the input document, and the negative subtree to not be matched. An example of such query is: $r/[a/X][\neg/b/X]$. We show the following (see Appendix).

**Theorem 4.4** It is undecidable, given a positive pattern $P$ without variables, and a recursion-free GXML schema $S$ with no data constraints and no external functions, but using patterns with negative sub-patterns, whether there exists an instance satisfying $P$ that is reachable in a valid run of $S$.

We next consider an extension of the Tree-LTL language. Recall that by definition, all free variables in the patterns of a Tree-LTL formula are universally quantified to yield the final Tree-LTL sentence. One might wonder if this restriction on the quantifier structure is needed for decidability of satisfaction for recursion-free GXML schemas. We next show that this is in fact the case. Specifically, let $\exists$Tree-LTL be defined the same as Tree-LTL, except that the free variables are quantified existentially in the end, yielding a sentence of the form $\exists X \xi(X)$.

**Theorem 4.5** It is undecidable, given a recursion-free GXML schema $S$ and a $\exists$Tree-LTL sentence $\varphi$, whether $S$ satisfies $\varphi$.

We finally consider the impact on decidability of allowing path quantifiers in the temporal property. To this end, we consider Tree-CTL properties and prove the following strong undecidability result (A is the universal quantifier and $\exists$ the existential quantifier on runs). It shows that allowing even a single path quantifier alternation leads to undecidability.

**Theorem 4.6** It is undecidable, given a positive pattern $P$ without variables and a recursion-free GXML schema $S$, if $S$ satisfies$^3$ $\Box \text{AXEG} (\neg P)$.

$^3$We assume a unique start state from which there is a transition to each initial instance of $S$. 
Decidability As promised, we now exhibit several useful verification tasks that remain decidable even for recursive GXML schemas. A recurring concern in verification is safety with respect to a specified property. Recall that reachability, and therefore safety, is undecidable by Theorem 4.1. We next provide a decidable sufficient condition for safety w.r.t. a Boolean combination of patterns.

Theorem 4.7 (Safety) It is decidable in co-NEXPTIME, given a GXML schema $S$ and a Boolean combination $\varphi$ of patterns, whether (i) all valid initial instances of $S$ satisfy $\varphi$, and (ii) for all valid instances $I$ and $J$ of $S$ such that $I \models J$, if $I \models \varphi$ then $J \models \varphi$.

Another practically significant problem is bounded reachability for given $k$, is it possible to reach in at most $k$ steps an instance satisfying a Boolean combination $\varphi$ of patterns? The following is shown similarly to the proof of Theorem 3.6.

Theorem 4.8 (Bounded reachability) It is decidable in 2NEXPTIME, given a GXML schema $S$, a Boolean combination of patterns $\varphi$, and a fixed integer $k$, whether there exists a prefix $I_0, \ldots, I_j$ of a valid run of $S$ such that $j \leq k$ and $I_j \models \varphi$. If $k$ is fixed, the complexity is NEXPTIME.

The dual of bounded reachability is bounded safety. for given $S$, $\varphi$ and $k$, is it the case that every instance of $S$ reachable in at most $k$ steps satisfies $\varphi$? Clearly, this is the case iff no instance satisfying $\neg \varphi$ can be reached in at most $k$ steps. Thus, bounded safety can be decided in co-2NEXPTIME (and co-NEXPTIME for fixed $k$).

5 Discussion

We studied the verification of an expressive set of properties for a large class of AXML systems. We aimed at providing a model capturing significant applications, while at the same time allowing for non-trivial verification tasks. Some of our choices include: unordered rather than ordered trees, set-oriented rather than bag semantics for trees, patterns with local existential quantification and without negated sub-patterns, and queries based on tree pattern matchings rather than more powerful computation. Despite the limitations, this goes beyond previous formal work on AXML, which considered only monotone systems [1]. Note that the use of guard conditions induces non-monotone behavior, since a call guard that is satisfied may later be invalidated when new data is received. Indeed, guards provide a powerful control mechanism, that allows simulating complex application workflows. Altogether, we believe the model captures a significant class of AXML services. Finally, the Tree-LTL language providing a novel coupling of temporal logic and tree patterns seems particularly well suited for expressing properties of the evolution of such systems.

Our results provide a tight boundary of decidability for verification of GXML systems. As a side effect, they also provide insight into the subtle interplay between the various features of GXML. Decidability for full verification holds for recursion-free GXML. While this may appear quite limited, applications often satisfy the recursion-free conditions required. Even in more complex applications that do not satisfy these conditions, one can isolate and verify recursion-free portions that are semantically significant. For instance, the Mail Order example can be made recursion-free by making MailOrder non-continuous. Intuitively, this corresponds to the processing of a single order, and properties of each such process can be verified. We also showed that more limited but useful verification tasks, such as bounded reachability and verifying sufficient conditions for safety, are decidable even for unrestricted GXML systems.

We conclude by discussing how our results can be extended to multi-peer systems, for which AXML was originally intended. The GXML model can simulate a multi-peer systems in a straightforward manner, by viewing the general system as a single GXML document with a separate portion assigned to each peer. This amounts to viewing the state of the multi-peer system as the product of the states of its components, in which each peer has access to its own state. Such a GXML system can easily simulate a multi-peer system under strong synchronicity assumptions ensuring that each function call causes simultaneous state transitions in the calling and receiving peers. This assumption can be immediately relaxed by introducing additional peers simulating communication channels, which weakens synchronicity by allowing arbitrary delays between state transitions in different peers. Simulating a finer-grained multi-peer model, with explicit messages and queues, requires an extension of our GXML model. This raises new interesting questions left for future work.
References


6 Appendix

6.1 Some definitions

A Guard AXML (GAXML) document is a tree whose internal nodes are labeled with tags in $\Sigma$ and whose leaves are labeled by either tags, function names, or data values. More formally, a GAXML tree is a triple $T = (N, E, \lambda)$ where $N \subseteq N$ is a finite set of nodes, $E \subseteq N \times N$ is a set of edges such that $(N, E)$ is a tree, and $\lambda : N \rightarrow \Sigma \cup F \cup D$ is a labeling function such that $\lambda(n) \in \Sigma$ for each nonleaf node $n$.

Patterns A tree pattern is a tuple $(M, G, \lambda_M, \lambda_G)$, where:

- $(M, G)$ is a tree with $M \subset N$,
- $\lambda_M : M \rightarrow \Sigma \cup F \cup D \cup \emptyset \cup \{\ast\}$ is a node labeling function such that $\lambda_M(n) \in \Sigma \cup \{\ast\}$ for every internal node $n$,
- $\lambda_G : G \rightarrow \{/\}$.

A pattern is a pair $((P_1, \ldots, P_n), cond)$, where each $P_i$ is a tree pattern and $cond$ is a Boolean combination of expressions $X = \alpha$ or $X \neq \alpha$, where $X \in \mathcal{V}$ and $\alpha \in \mathcal{V} \cup \mathcal{D}$. In particular $cond$ could include joins of the form $X = Y$.

Let $P$ be a tree pattern and $T$ a tree. A matching of $P$ into $T$ is a mapping $\mu$ from the nodes of $P$ to the nodes of $T$ such that: (i) the root of $P$ is mapped to the root of $T$, (ii) $\mu$ interprets `/` as child and `//` as descendant, (iii) $\mu$ preserves the label (with `*` acting as a wildcard), (iv) nodes labeled with variables are mapped to data values. Let $Q = ((P_1, \ldots, P_n), cond)$ be a pattern. A matching of $Q$ into a forest $F$ is a mapping $\mu$ that is a matching of each $P_i$ into some tree of $F$, and for which $cond$ is satisfied.

We say that a pattern $Q$ holds in a forest $F$ iff there exists at least one matching of $Q$ into $F$. We then say that $Q(F)$ is true, otherwise it is false. This definition extends to Boolean combination of patterns by replacing each pattern $Q$ by $Q(F)$. In particular this means that the patterns are matched independently of each other: If a variable $X$ occurs in two different patterns $Q$ and $Q'$ of the Boolean combination, then it is treated as quantified existentially in $Q$ and independently quantified in $Q'$.

6.2 Mail Order Example

We provide a more complete specification for our running MailOrder example. The purpose of this GAXML system is to process mail orders. The system has access to a Catalog, providing product and price information. A new mail order is initiated by an external call !MailOrder. The processing of a mail order follows this simple workflow:

1. Receive an order from a customer $Cname$ for a product $Pname$. The order is given a unique identifier $OrderID$ (uniqueness is enforced by the data constraint specified further).
2. If the product is available, initiate processing a bill by calling the internal function $Bill$.
3. To process a bill, send an invoice to the customer, modeled by a call to the external function $Invoice$. This returns a $Payment$ for $Pname$ in the amount found under $Amount$. This completes the processing of the bill. $Pname$ and $Amount$ are returned to the calling MailOrder as the answer to the call $!Bill$.
4. If the payment is correct (the catalog price of the product $Pname$ is the paid $Amount$) then deliver the product by calling the external function $Deliver$. Otherwise reject the order by calling the external function $Reject$.

We now provide more details on the specification (for convenience, some aspects already described in the main text are repeated here). An initial instance of the system has the shape shown in Figure 1. The DTD enforces the specified shape, and also that of the results to external function calls, described later. The uniqueness of mail order IDs is enforced by the data constraint consisting of the negation of the following pattern:

We next provide the specifications of the functions.

$MailOrder$ : external, continuous. Its call guard is $true$ and argument query empty. Its result has the following shape, enforced by the DTD:

$$
\text{MailOrder} \quad | \quad \text{Bill} \quad | \quad \text{Deliver} \quad | \quad \text{Reject}
$$

$$
\text{Order-Id} \quad | \quad \text{Cname} \quad | \quad \text{Pname} \quad | \quad \text{Bill} \quad | \quad \text{Deliver} \quad | \quad \text{Reject}
$$

We provide the specifications of the functions.
Bill : internal, non-continuous. Its call guard, checking that the ordered product is available, is the following:

\[
\begin{array}{c}
\text{Main} \\
\text{MailOrder} \\
\text{Product} \\
\text{Pname} \\
\text{X} \\
\end{array}
\quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{\{\text{Process-bill}\}}{\text{Pname}} \\
\begin{array}{c}
\text{Pname} \\
\text{Amount} \\
\text{X} \\
\end{array}
\]

Its argument query is:

\[
\begin{array}{c}
\text{Main} \\
\text{Catalog} \\
\text{Product} \\
\text{Pname} \\
\text{X} \\
\end{array}
\quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{\{\text{Process-bill}\}}{\text{Pname}} \\
\begin{array}{c}
\text{Pname} \\
\text{Price} \\
\text{X} \\
\end{array}
\]

The return guard and query (also given in Example 2.2) are the following:

\[
\begin{array}{c}
\text{Payment} \\
\text{Pname} \\
\text{Amount} \\
\text{X} \\
\end{array}
\quad \frac{\{\text{Process-bill}\}}{\text{Return guard}} \\
\begin{array}{c}
\text{Return query} \\
\text{X} \\
\text{Y} \\
\end{array}
\]

Invoice : external, non-continuous. Its call guard is \textit{true}. We omit (as for the other external functions) the specification of its argument query. The answer it returns is of the following form (which can be enforced by the DTD):

\[
\begin{array}{c}
\text{Payment} \\
\text{Pname} \\
\text{Amount} \\
\text{X} \\
\end{array}
\quad \frac{\text{dom}}{\text{dom}}
\]

Deliver : external, non-continuous. Its call guard is

\[
\begin{array}{c}
\text{Main} \\
\text{Catalog} \\
\text{Product} \\
\text{Pname} \\
\text{X} \\
\end{array}
\quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{\{\text{Process-bill}\}}{\text{Pname}} \\
\begin{array}{c}
\text{Pname} \\
\text{Price} \\
\text{X} \\
\end{array}
\]

The result consists of a single node labeled \textit{Delivered} (this can be enforced by the DTD).

Rejected : external, non-continuous. Its call guard is the following:

\[
\begin{array}{c}
\text{Main} \\
\text{Catalog} \\
\text{MailOrder} \\
\text{Product} \\
\text{Pname} \\
\text{X} \\
\end{array}
\quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{a_{\text{Bill}}}{\text{Payment}} \quad \frac{\{\text{Process-bill}\}}{\text{Pname}} \\
\begin{array}{c}
\text{Pname} \\
\text{Amount} \\
\text{X} \\
\end{array}
\]

The result consists of a single node labeled \textit{Rejected} (this can also be enforced by the DTD).

This completes the specification of the Mail Order GXML system.

Now consider again the Tree-LTL properties in Figure 5. The first property (every mail order is eventually delivered or rejected) is satisfied for the above specification. Consider the second property (every product for which the correct amount has been paid is eventually delivered). Surprisingly, this property is false. This is due to a subtle bug: the specification allows a customer to pay for a different product than the one ordered. This bug could be fixed with the addition of the data constraint consisting of the negation of the following pattern:

\[
\begin{array}{c}
\text{Return guard} \\
\text{Return query} \\
\end{array}
\quad \frac{\text{Return guard}}{\text{Return query}}
\]

\[
\begin{array}{c}
\text{Payment} \\
\text{Pname} \\
\text{Amount} \\
\text{X} \\
\end{array}
\quad \frac{\text{dom}}{\text{dom}}
\]

6.3 Upper bound proofs for recursion-free systems

**Proposition 3.2** Let \( S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta) \) be a recursion-free GXML schema. There exists a non-negative integer \( k \), exponential in \(|\Phi_{\text{int}} \cup \Phi_{\text{ext}}|\), such that all runs of \( S \) reach a blocking instance in at most \( k \) transitions.

**Proof:** Let \( \Phi = \Phi_{\text{int}} \cup \Phi_{\text{ext}} \) and \( \kappa = |\Phi| \). Let \( G \) be the call graph of \( S \). By assumption \( G \) is acyclic. Let \( G_0 \) be the set of functions \( f \in \Phi \) such that the corresponding node has no incoming edge in \( G \). The depth of a function \( f \) of \( \Phi \) is the longest distance between the node representing \( f \) in \( G \) and a node of \( G_0 \). We show by induction of \( i \) that a function \( f \) of depth \( i \) can be called at most \((2 \cdot \kappa)^i\) times. For \( i = 0 \) this is clear, the function may be called only if it is present in the initial instance and, because \( S \) is recursion-free and the initial instance is a tree, it can only occur once in the initial instance. For arbitrary \( i \), let \( G_f \) be the set of parents of \( f \) in \( G \). By induction a function \( g \) of \( G_f \) can be called at most \((2 \cdot \kappa)^{i-1}\) times.
times. Because $S$ is recursion-free, one execution of $g$ may produce at most two direct executions of $f$ (one generated by the input query, one by the output query). Hence $f$ is executed at most $2 |G_f|(2 \cdot \kappa)^{-1} \leq (2 \cdot \kappa)^k$ times as $|G_f| \leq \kappa$. The depth of $G$ being bounded by $\kappa$, each function is eventually executed at most $(2 \cdot \kappa)^k$ times, hence the bound on the length of the run. \hfill \Box

**Proposition 3.3** Given a pre-run $\rho = I_0, \ldots, I_k$ of $S$ and a Tree-LTL sentence $\varphi$, one can check whether $\rho$ satisfies $\varphi$ using a non-deterministic algorithm in time $O(|\rho|^{|\varphi|})$.

**Proof:** Let $P$ be the set of tree patterns used in $\varphi$. For each $m \in [0, k]$, let $\sigma_m$ be the truth assignment on $P$ such that $\sigma_m(P) = 1$ iff $I_m \models P$ (note that the latter can be checked in time exponential in $P$). Let $A_\varphi$ be the Büchi automaton for the formula $\varphi$ where the tree patterns are replaced by distinct propositions (also denoted by $P$ by slight abuse), and whose alphabet consists of the truth assignments for $P$. The standard construction of $A_\varphi$ produces an automaton whose number of states is exponential in $|\varphi|$. Recall that (by definition of $A_\varphi$) $\rho \models \varphi$ iff $A_\varphi$ accepts the infinite word $\sigma_0, \sigma_1, \ldots, \sigma_k, \ldots$, i.e. iff $A_\varphi$ goes infinitely often through an accepting state. A simple pumping argument shows that this happens iff an accepting state can be reached twice from a state reached under input $\sigma_0, \sigma_1, \sigma_k$ by reading again $\sigma_k$ at most $2 \cdot |A_\varphi|$ times. This yields the desired non-deterministic algorithm taking time $O(|\rho|^{|\varphi|})$. \hfill \Box

**Theorem 3.6** It is decidable in co-2NEXPTIME, given a recursion-free GXML schema $S$ and a Tree-LTL sentence $\varphi$, whether each valid run of $S$ satisfies $\varphi$.

We next outline the main points of the proof.

Let $S = (\Phi_{in}, \Phi_{ext}, \Delta)$ be a recursion-free GXML schema and $\varphi$ a Tree-LTL sentence of the form $\forall X \psi(\bar{X})$. Clearly, $S \models \varphi$ iff there is no valid run of $S$ that satisfies $\exists X \psi(\bar{X})$. Let $D_X$ be an arbitrary subset of $X$ with as many elements as variables in $\bar{X}$. Clearly, the above is equivalent to the following: there is no valid run $\rho$ of $S$ with domain $D \supseteq D_X$ and no mapping $h$ from $\bar{X}$ to $D_X$ such that $\rho$ satisfies $\xi = \neg \psi(h(\bar{X}))$ (recall that $\psi(h(\bar{X}))$ is obtained from $\psi$ by replacing, for each pattern in $\psi$ for which $Y \in \bar{X}$ is a free variable, the label $Y$ by $h(Y)$). Thus, the question of whether $S \models \varphi$ is reduced to a satisfiability problem. Decidability is shown by proving a small model property. Specifically, we show the following key lemma.

**Proposition 3.4** There exists a computable function $f$ with the following property. If there exists a valid pre-run of $S$ satisfying $\xi$ then there exists a valid pre-run of the same length satisfying $\xi$, such that each of its instances has size bounded by $f(\xi, S)$.

Clearly, Proposition 3.4 establishes Theorem 3.6. Indeed, as will be seen, $f$ is doubly exponential in $S$ and $\xi$, which will yield a complexity of 2NEXPTIME for checking the existence of a valid run satisfying $\xi$ (so violating $\varphi$). This establishes the co-2NEXPTIME upper bound for checking satisfaction of $\varphi$.

The main idea of the proof is as follows. Let $I_0, \ldots, I_k$ be a valid pre-run of $S$ satisfying $\xi$. We construct another valid pre-run $R_0, \ldots, R_k$ such that for each $m \in [0, k]$, $R_m$ is a sub-instance of $I_m$ whose size can be statically bounded, and $R_m$ and $I_m$ satisfy exactly the same patterns used in $\xi$. The idea is to make sure that each $R_m$ contains witnesses for all patterns in $\xi$ satisfied by $I_m$, and also that it can mimic the transitions in the original run by keeping the "skeleton" of $I_m$ (all paths from roots to nodes labeled with function symbols $?f$ or $a_f$) and also witnesses required to make the appropriate guards true. Satisfaction of the DTD must also be ensured, which requires additional witnesses. The construction is done in two passes: first, the needed witnesses are collected starting from $I_0$ and backward to $I_0$. Then, the actual pre-run $R_0, \ldots, R_k$ is generated starting from the sub-instance of $I_0$ containing the collected witnesses, by mimicking the transitions in the original run.

In order to establish Proposition 3.4 we first show several auxiliary results. Note that, for the proof, $\Delta$ can be assumed to consist only of a DTD, since the data constraints can be absorbed into the property $\varphi$ to be verified.

We use the following terminology. Let $I = (T, eval)$ be an instance of $S = (\Phi_{in}, \Phi_{ext}, \Delta)$. A sub-instance of $I$ is an instance $J = (T', eval')$ of $S$ such that (i) each tree $T'$ in $T'$ is a prefix of some tree $T$ in $T$, (ii) $T'$ includes all nodes in $T$ labeled by function symbols $?f$ and $a_f$, and $eval'$ maps each node $n$ labeled $?f$ to the tree in $T'$ that is a prefix of $eval(n)$. We denote by $J \subseteq I$ the fact that $J$ is a sub-instance of $I$. Note that $eval'$ is uniquely determined by $T'$

\footnote{A tree $T'$ is a prefix of a tree $T$ if $T'$ is a subgraph of $T$ and for each node $x$ in $T'$, all nodes on the path from $x$ to the root in $T$ are also in $T'$.}
An important property of a subinstance is that it preserves the false patterns: If a pattern does not hold in \( I \) then it doesn’t hold in any of its subinstances.

The next result shows how we can “propagate backwards” sub-instances throughout a run. Note that the lemma does not assume non-reversiveness.

**Lemma 6.1** Let \( S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta) \) be a GAXML schema, \( I \) and \( I' \) instances of \( S \) such that \( I \vdash I' \), and let \( K \subseteq I' \). Then there exists \( \text{Pre}(K) \subseteq I \) and \( K' \subseteq I' \) such that \( \text{Pre}(K) \vdash K' \), \( K \subseteq K' \), and \( |\text{Pre}(K)| \leq (d \cdot \gamma(f)) + d \cdot |\text{Body}| \cdot |K| \) where \( d \) is the maximum depth of a tree satisfying \( \Delta \), \( \gamma(f) \) is the size of the guard of \( f \), and \( \text{Body} \) is the body of an argument or return query in \( S \).

**Proof:** We do a case analysis on the transition \( I \vdash I' \). Suppose first that \( I' \) is obtained from \( I \) as a result of a function call \( !f \) at a node \( n \). Let \( \gamma(f) \) be the guard of \( f \). For each pattern \( P \) occurring in \( \gamma(f) \), there exists a matching of \( P \) into \( (I, n) \), and let \( G \) be the forest induced by all the nodes in the images of some \( \mu_P \) together with their ancestors. Note that the size of \( G \) is bounded by \( d \cdot \gamma(f) \).

If \( f \) is an external function, then \( \text{Pre}(K) \) consists of \( K \) together with \( K' \), the label of \( n \) changed from \( ?f \) to \( !f \). Suppose \( f \) is an internal function. Let \( T \) be the tree in \( I' \) rooted at \( n \) resulting from the call. Recall that \( T \) consists of \( \alpha \) with subtrees resulting from the evaluation of the argument query of \( f \), \( \text{Body} \to \text{Head} \) on \( (I, n) \). For each tree \( H \) in \( \text{Head} \), let \( c \) be its constructor node and \( H_c \) the corresponding subtree. For each matching \( \mu \) of \( \text{Body} \) into \( (I, n) \) we denote by \( \mu(H_c) \) the set of nodes of \( I' \) that are induced by this matching. Let \( \mathcal{M} \) be the set of all \( \mu \) for which \( \mu(H_c) \) intersects \( K \) for some \( H \) in \( \text{Head} \). Then the nodes of \( \text{Pre}(K) \) are those of \( G \) together with those belonging to both \( K \) and \( I' \), and those occurring in \( \{ \mu(\text{Body}) \mid \mu \in \mathcal{M} \} \), together with their ancestors. Note that \( |\text{Pre}(K)| \leq |K| + d \cdot (\gamma(f)) + d \cdot |\text{Body}| \cdot |K| \leq (d \cdot \gamma(f)) + (d \cdot |\text{Body}| \cdot |K|) \) to see that \( \text{Pre}(K) \) satisfies the other conditions of the lemma, note first that \( \text{Pre}(K) \) contains \( n \) with label \( !f \) (the same as in \( I \) and \( \gamma(f) \) holds in \( \text{Pre}(K), n \)). Thus, there exists \( K' \) such that \( \text{Pre}(K) \vdash K' \) and \( K' \) is obtained from \( \text{Pre}(K) \) by a call to \( f \) at node \( n \). If \( f \) is an external function, \( K \subseteq K' \) by construction. If \( f \) is an internal function, \( K' \) is obtained from \( \text{Pre}(K) \) by evaluating \( \text{arg}(f) = \text{Body} \to \text{Head} \) on \( \text{Pre}(K), n \). Since by construction all matchings of \( \text{Body} \) into \( (I, n) \) in \( \mathcal{M} \) are also matchings in \( \text{Pre}(K), n \), it easily follows that \( K \subseteq K' \) (modulo node renaming). Since \( \text{Pre}(K) \subseteq I \) and \( I \vdash I' \) it also follows that \( K' \subseteq I' \).

Next, suppose \( I' \) is obtained from \( I \) by the return of the result of a function call \( ?f \) at node \( n \). Suppose \( f \) is an external function. Then \( \text{Pre}(K) \) is the smallest sub-instance of \( I \) containing \( K \) from which the subtrees belonging to the result of the call are deleted. If \( f \) is an internal function, \( \text{Pre}(K) \) is obtained similarly to the above. In this case again, \( |\text{Pre}(K)| \leq |K| + d \cdot \gamma(f) + d \cdot |\text{Body}| \cdot |K| \) where \( \gamma(f) \) is the return guard of \( f \) and \( \text{Body} \) is the body of the return query of \( f \), so \( |\text{Pre}(K)| \leq d \cdot \gamma(f) + (d \cdot |\text{Body}| \cdot |K|) \). The proof that \( \text{Pre}(K) \vdash K' \) where \( K \subseteq K' \subseteq I' \) is similar to the above. \( \square \)

In constructing our "small" run, we will need to enforce validity of the instances wrt \( \Delta \). To this end, we use the notion of "completion" of an instance. Let \( J \) be a sub-instance of \( I \). A completion \( J \) of \( I \) wrt \( I \) and \( \Delta \) is defined as follows.

**Lemma 6.2** Let \( S \) be a GAXML schema. Suppose \( I \) is an instance of \( S \), \( I \models \Delta \) is a sub-instance of \( I \), and \( J \) is a completion of \( J \) wrt \( I \) and \( \Delta \). Then for every instance \( L \) such that \( J \subseteq L \subseteq I \), if \( n \) is a node in \( J \), then the set of children of \( n \) in \( L \) satisfies \( \Delta \) (in particular, \( J \models \Delta \)). Furthermore, \( |J| \leq d \cdot (a \cdot \max(\Delta))^{d} \cdot |J| \), where \( d \) is the maximum depth of a tree satisfying \( \Delta \) and \( a \) is the size of the alphabet of \( \Delta \).

**Proof:** Suppose \( J \subseteq L \subseteq I \) and \( J \) is a node in \( J \). Consider the children of \( n \) in \( L \). By construction, for each label \( b \), the number of children of \( n \) with label \( b \) in \( L \) is either the same as in \( I \) or lies between \( \max(\Delta) \) and the number of such children in \( I \) (and similarly for nodes labeled with data values). In either case, for \( k \leq \max(\Delta) \), \( b \geq k \) holds in \( I \) if it holds in \( I \) for the children of \( n \). Since \( I \models \Delta \), it follows that \( \Delta \) is satisfied by the children of \( n \) in \( L \). Finally, the bound on \( J \) is immediate. \( \square \)
Proof of Proposition 3.4 We are now ready to complete the proof of Proposition 3.4. Let \( \rho = I_0, \ldots, I_k \) be a valid pre-run of \( S \) satisfying \( \xi \). We construct a valid pre-run \( R_0, \ldots, R_k \) of bounded size such that for all \( m \in [0, k] \), \( I_m \) and \( R_m \) satisfy exactly the same patterns occurring in \( \xi \). Since \( I_1, \ldots, I_k \) satisfies \( \xi \), so does \( R_1, \ldots, R_k \).

Let \( \mathcal{P} \) be the set of patterns occurring in \( \xi \). For each \( m \) in \([0, k]\), and pattern \( P \in \mathcal{P} \) that holds in \( I_m \), let \( \sigma_{P, m} \) be one matching of \( P \) into \( I_m \), and let \( \text{Match}_m(\mathcal{P}) \) be the image of \( \{ \sigma_{P, m} \mid P \in \mathcal{P}, I_m \models P \} \). The skeleton of \( \rho \) is the set of all nodes occurring on a path from root to a node labeled with a function symbol \( f \) or \( a_f \) in some \( I_m \), \( 0 \leq m \leq k \). We define by backward induction valid sub-instances \( J_m \) of \( I_m \) as follows. For the basis, consider \( m = k \). Recall that \( I_k \) is blocking. For each node \( n \) in \( I_k \) labeled by a function call \( I_k \), and each pattern \( P \) in \( \gamma(f) \) that matches into \( (I_k, n) \), let \( \sigma_P \) be such a matching. Let \( G \) be a set of nodes in the image of all such matchings. Let \( J_k \) be the minimum sub-instance of \( I_k \) that includes \( G \), all nodes of \( I_k \) that belong to the skeleton of \( \rho \), and all nodes in \( \text{Match}_k(\mathcal{P}) \). Let \( J_k \) be a completion of \( J_k \) wrt \( I_k \) and \( \Delta \).

For the inductive step, let \( m < k \). Let \( \text{Pre}(J_{m+1}) \) be constructed from \( J_{m+1} \) as in Lemma 6.1. Next, let \( J_m \) be the minimum sub-instance of \( I_m \) containing \( \text{Pre}(J_{m+1}) \), the nodes of \( I_m \) that belong to the skeleton of \( \rho \), and the nodes in \( \text{Match}_m(\mathcal{P}) \). Finally, let \( J_m \) be a completion of \( J_m \) wrt \( I_m \) and \( \Delta \).

We next define by forward induction the desired valid pre-run \( R_0, \ldots, R_k \), starting with \( R_0 = J_0 \). As we shall see, \( J_m \subset R_m \subset I_m \) for \( 0 \leq m \leq k \) and \( R_m \models \Delta \). The basis \( (R_0 = J_0) \) is clear. Let \( 0 \leq m < k \) and suppose \( R_m \) has been defined, \( R_m \) satisfies \( \Delta \), and \( J_m \subset R_m \subset I_m \). By construction, \( \text{Pre}(J_{m+1}) \subset J_m \), so \( \text{Pre}(J_{m+1}) \subset R_m \). By Lemma 6.1, \( \text{Pre}(J_{m+1}) \models K' \) where \( J_{m+1} \subset K' \subset I_{m+1} \). Since \( \text{Pre}(J_{m+1}) \subset R_m \subset I_m \), it follows that \( R_m \models R_{m+1} \) where \( K' \subset R_{m+1} \subset I_{m+1} \), and the transition \( R_m \models R_{m+1} \) results from the same function call or result return as in \( I_m \models I_{m+1} \). The transition is uniquely determined, except in the case of the return of the result of an external call. In this case, consider the forest \( F \) which is the result of the same function call in \( I_{m+1} \). Let \( R_{m+1} \) be obtained from \( R_m \) by returning as answer to the external call \( F \cap J_{m+1} \). In all cases, since \( J_{m+1} \subset K' \), we have the desired inclusions \( J_{m+1} \subset R_{m+1} \subset I_{m+1} \).

To see that \( R_{m+1} \models \Delta \), consider the possible transitions from \( R_m \) to \( R_{m+1} \). Suppose \( R_{m+1} \) is obtained from \( R_m \) by a function call to \( f \). If \( f \) is external, \( \Delta \) is clearly satisfied. If \( f \) is internal, note that, since \( R_m \models \Delta \), the only violation of \( \Delta \) in \( R_{m+1} \) could occur if the number of trees in the answer to the argument query of the call on \( R_m \) is disallowed by \( \Delta \) under root \( a_f \). However, this cannot happen by Lemma 6.2, since \( J_{m+1} \subset R_{m+1} \) and the root belongs to \( J_{m+1} \).

Now suppose \( R_{m+1} \) is obtained from \( R_m \) by the return of the result of a call to a function \( f \). If \( f \) is internal, the argument is similar to the above (we use here the fact that the root under which the result of the function call is returned in \( I_{m+1} \) is part of the skeleton of \( \rho \) so belongs to \( J_{m+1} \)). Suppose \( f \) is external. Recall that by construction, the answer to the external call consists of sibling subtrees of \( J_{m+1} \) sitting under some node \( n \), and each tree in the answer satisfies \( \Delta \) (because \( I_{m+1} \models \Delta \)). \( R_{m+1} \) may contain additional sibling subtrees under \( n \) that are not in \( J_{m+1} \) because they were already in \( R_m \), and each satisfies \( \Delta \). Since \( J_{m+1} \subset R_{m+1} \), and \( n \) is in \( J_{m+1} \), the set of children of \( n \) also satisfy \( \Delta \), by Lemma 6.2. Thus, \( R_{m+1} \models \Delta \). This completes the induction.

Clearly, for each \( m \in [0, k] \), \( R_m \) and \( I_m \) satisfy exactly the same patterns in \( \mathcal{P} \), because \( J_m \subset R_m \subset I_m \) and each \( P \in \mathcal{P} \) that holds in \( I_m \) also has a match in \( J_m \), so in \( R_m \). Conversely, if \( P \) does not hold in \( I_m \) it cannot hold in \( R_m \). Finally, \( R_k \) is blocking because \( I_k \) is blocking and \( J_k \) and \( I_k \) satisfy exactly the same patterns occurring in the call guards.

We now provide a bound for the pre-run \( R_0, \ldots, R_k \). We denote by \( s \) the size of \( S \) and by \( x \) the size of \( \xi \). Recall that \( d \) is the depth of all trees that are valid for \( \Delta \) and that \( d \leq s \). Recall also that \( g \) is the maximum size for a guard and that \( g \leq s \). From the above it follows that:

- \( k \) is exponential in \( s \) (Proposition 3.2).
- The skeleton \( sk(\rho) \) of \( \rho = I_0, \ldots, I_k \) is bounded by \( k \cdot d \cdot 2^k \). To see this notice that a run of length \( k \) can call at most \( k \) functions. Hence an instance of this run has at most \( k \) nodes labeled \( a_f \) and \( k \) nodes labeled \( ?f \). Each such node has at most \( d \) ancestors and there are \( k \) instances. Hence \( sk(\rho) = O(s \cdot k^2) \).
- The size of \( J_k \) is bounded by \( |sk(\rho)|+d \cdot g \cdot 2^k \). The term \( d \cdot g \cdot 2^k \) bounds the size of \( G \) (we need to consider a most \( 2 \cdot k \) guards) and \( d \cdot |\xi| \) bounds the size of \( \text{Match}_k(\mathcal{P}) \). Thus, \( |J_k| = O(s^2 \cdot x \cdot k^2) \).
• By Lemma 6.2, \(|\mathcal{J}| = O(s^{2s+1} \cdot |J|)\), so \(|\mathcal{J}_k| = O(k^2 \cdot x \cdot s^{2s+3})\).

• Consider \(\mathcal{J}_m\) for \(m < k\). By construction, 
  \(|\mathcal{J}_m| \leq |\text{Pre}(\mathcal{J}_{m+1})| + |sk(\rho)| + |\text{Match}_m(\mathcal{P})|\).

By Lemma 6.1, \(|\text{Pre}(\mathcal{J}_{m+1})| = O(s^2 \cdot |\mathcal{J}_{m+1}|)\). Also, 
\(|sk(\rho)| = O(s \cdot k^2)\) (see above) and  
\(|\text{Match}_m(\mathcal{P})| = O(s \cdot x)\). It follows that 
\(|\mathcal{J}_m| = O((s^{2s+1} + s \cdot k^2 + s \cdot x) = O(k^2 \cdot x \cdot s^{2s+3} \cdot |\mathcal{J}_{m+1}|)\).

From the above, it follows that 
\(|\mathcal{J}_0| = O((k^2 \cdot x \cdot s^{2s+3})^k \cdot |\mathcal{J}_k|) = O((k^2 \cdot x \cdot s^{2s+3})^k \cdot k^2 \cdot x \cdot s^{2s+3}) = O(k^{2k+1} \cdot x^{k+1} \cdot s^{(2s+3)(k+1)})\).

Thus, \(\mathcal{J}_0\) is doubly exponential with respect to \(S\) and \(\xi\). Now consider the pre-run \(R_0, \ldots, R_k\). At each transition \(R_m \vdash R_{m+1}\), the instance \(R_m\) can increase by at most \(|\mathcal{J}_0|^v \cdot h\), where \(v\) is the maximum number of variables in the head of a query of \(S\), and \(h\) is the maximum size of a query head. Recall that by construction, the result of an external call is bounded by the maximum size of \(|\mathcal{J}_m|, m \in [0, k]\), to which the bound established above for \(|\mathcal{J}_0|\) applies. Thus, each \(R_m\) remains doubly exponential in \(S\) and \(\xi\). This completes the proof of Proposition 3.4.

In summary, a 2\textsc{NExpTime} decision procedure for satisfiability of \(\xi\) by \(S\) is the following:

1. Guess \(D_\xi\) and the valuation \(h\) of \(\text{X}\) into \(D_\chi\); construct the formula \(\xi\).

2. Guess an initial instance \(R_0\) of a valid pre-run of \(S\), of size doubly exponential in \(S\) and \(\xi\).

3. Generate non-deterministically a valid pre-run \(R_0, \ldots, R_k\) of \(S\); in the case of external function calls, guess an arbitrary answer whose size is bounded by the bound shown above for \(|\mathcal{J}_0|\). A blocking instance \(R_k\) is guaranteed to be reached after a number of transitions exponential in \(S\).

4. Check that \(R_0, \ldots, R_k\) satisfies \(\xi\).

Note that (4) remains in 2\textsc{NExpTime} by Proposition 3.3. This completes the proof of Theorem 3.6. □

Corollary 3.8 The following are decidable in co-2\textsc{NExpTime} for a recursion-free GAXML schema \(S = (\Phi_{\int}, \Phi_{\ext}, \Delta)\):

- Successful termination: each run of \(S\) ends in a blocking instance with no running function call.
- Typechecking: for every run of \(S\), if the initial instance satisfies \(\Delta\), then every instance in the run satisfies \(\Delta\).

Proof: Successful termination can be reduced to satisfaction of a Tree-LTL sentence by a recursion-free system. For successful termination, the property to be verified is 
\(F [\alpha \land \bigwedge_{f \in \Phi_{\int} \cup \Phi_{\ext}} \neg \gamma(f)]\) where \(\alpha\) is a formula stating that no function symbol \(?f\) is present, and each \(\gamma(f)\) is obtained from the guard \(\gamma(f)\) by replacing the label \(self\) by \!\!f. This uses the fact that, in a tree without function calls, the DTD of \(\Delta\) does not allow multiple occurrences of nodes labeled !f (thus, the relative pattern \(\gamma(f)\) can be turned into the pattern \(\gamma(f)\) without any loss). Also note that, since the initial instance of a run consists of a single tree, every reachable instance without running function calls is also a single tree.

For typechecking, the proof is analogous to that of Lemma 3.4. Suppose \(\Delta\) consists of a DTD \(\Delta'\) and a data constraint \(\psi\). We first typecheck \(\Delta'\); we show that whenever \(\rho = I_0, \ldots, I_{k}\) is a prefix of a run of \(S\) such that \(I_0\) satisfies \(\Delta\), \(I_0\) satisfies \(\Delta'\). Suppose, to the contrary, that there exists \(\rho = I_0, \ldots, I_k\) such that \(I_0\) is an initial instance (satisfying \(\Delta\), \(I_m \vdash I_{m+1}\), and \(I_k\) violates \(\Delta'\). We construct a sequence \(\rho' = R_0, \ldots, R_k\) with the same properties, such that the size of \(\rho'\) is statically bounded. The construction is the same as in the proof of Lemma 3.4 (for \(\mathcal{P} = \emptyset\)), with two modifications. First, \(J_k\) is defined as follows. Since \(I_k\) violates \(\Delta'\), there is a node \(n\) of \(I_k\) whose children violate \(\Delta'\). Let \(J_k\) be the minimum sub-instance of \(I_k\) containing \(n\) together with \(sk(\rho)\), and let \(\tilde{J}_k\) be the completion of \(\tilde{J}_k\) wrt \(I_k\) and \(\Delta'\). Note that \(\tilde{J}_k\) still violates \(\Delta'\). The inductive construction of \(J_m\) for \(m < k\) proceeds as in Lemma 3.4, except that \(J_k\) also includes witnesses for the patterns of \(\psi\) satisfied by \(I_0\). Next, \(\rho' = R_0, \ldots, R_k\) is constructed from \(J_0\) as in Lemma 3.4. Note that \(J_k \subseteq R_k \subseteq I_k\), and \(R_k\) still violates \(\Delta'\) for the same node \(n\) as \(J_k\) and \(I_k\). Clearly, \(\rho'\) has the desired properties and has size doubly exponential in \(S\). This shows that checking the existence of a violation of typechecking wrt the DTD \(\Delta'\) can be done in 2\textsc{NExpTime}, so typechecking wrt \(\Delta'\) is in co-2\textsc{NExpTime}. Now consider \(\Delta\). If the answer to the above is negative (there is a violation of \(\Delta'\)) then we are done (\(\Delta\) is also violated). Otherwise, let \(S' = (\Phi_{\int}, \Phi_{\ext}, \Delta')\), and check that every valid pre-run of \(S'\) satisfies the Tree-LTL property \(\psi \rightarrow G \psi\). This can be done in co-2\textsc{NExpTime} by Theorem 3.6. In summary, typechecking is decidable in co-2\textsc{NExpTime}. □
6.4 Lower bound proof

We now prove that the complexity bound of Theorem 3.6 is tight.

Proof of Proposition 3.5 Let \( M \) be a non-deterministic Turing Machine running in time \( 2^2n \) on inputs of size \( n \).

Let \( w \) be a string of length \( n \). We construct a tree-LTL sentence \( \varphi \) and a recursion-free GXML service \( S \) such that \( \varphi \) holds on all runs of \( S \) iff \( M \) rejects \( w \).

The general idea is that any initial instance of \( S \) should code a valid run of \( M \) on \( w \). If this is not the case, there will be a run of \( S \) that will violate \( \varphi \).

**Encoding the configurations of \( M \).** The initial instance of \( S \) roughly looks as follows (for technical reasons there will be some extra features that we will introduced when needed):

![Diagram of tree-LTL configuration encoding]

Each tree rooted by a symbol \( R \) is expected to code a configuration of \( M \). Data values (not depicted above) are attached to the \( N \) and \( P \)-nodes in order to form a successor relation between the leaves (recall that our trees are unordered). The label of each leaf (the round circles in the figure above) is a tape symbol of \( M \). In each \( R \)-subtree, the sequence of labels of the leaves induced by this successor relation codes a configuration of \( M \). The \( U \)-nodes are here for technical reasons that will become apparent later. Note that the global structure without the data values can easily be enforced using a DTD.

**Enforcing a successor relation.** This is the difficult part of the proof, the rest being standard. We assume that each \( N \)-node and each \( P \)-node has a child containing a data value. We use these data values in order to code the successor relation. The idea is that if a data value of an \( N \)-node appears in a \( P \)-node then the corresponding branches are successive. More precisely we derive a directed graph \( G \) as follows. The vertices of \( G \) are all the data values of \( N \)-nodes. There is an arc \( (\alpha, \beta) \) in \( G \) iff \( \alpha \) and \( \beta \) are distinct data values of \( N \)-nodes \( x \) and \( y \), such that the data value of the \( P \)-child of \( y \) is \( \alpha \). We specify \( S \) such that we can detect whether \( G \) induces a sequence long enough in each \( R \)-subtree for coding a configuration of \( M \) and such that \( G \) also induces a sufficiently long sequence of configurations. In particular, for coding the run of \( M \), \( G \) must contain a sequence of length \( 2^{2n} \cdot 2^{2^{n+1}} = 2^{2^{2n+1}} \).

For technical reasons, we will distinguish between three kinds of \( N \)-nodes, \( N_{\text{beg}}, N_{\text{last}}, \) and \( N_{\text{inner}} \). We ensure that each \( R \)-subtree contains exactly one occurrence of \( N_{\text{beg}} \), one occurrence of \( N_{\text{last}} \), and that all the remaining \( N \)-nodes are \( N_{\text{inner}} \). This can be done by adding one more child to the \( N \)-nodes with a specific finite label for each kind. The constraints above can then be enforced using a DTD. We will use these nodes to code respectively the first, the last and the other elements of the sequence induced by \( G \) on each \( R \)-tree. In the patterns depicted in the following we will use \( N_{\text{beg}}, N_{\text{last}} \) and \( N_{\text{inner}} \) as labels instead of explicitly referring to the label of their child. To simplify further the presentation we will also use in the patterns the label \( R_{\text{beg}}, R_{\text{last}} \) and \( R_{\text{inner}} \) for \( P \)-nodes whose parent is respectively a \( N_{\text{beg}}, N_{\text{last}} \) and \( N_{\text{inner}} \) node. Similarly we mark one of the \( R \)-subtree for the initial configuration, one of the \( R \)-subtree for the final configuration and denote them by \( R_{\text{beg}} \) and \( R_{\text{last}} \). Again this can be enforced by a DTD.

With patterns it is easy to enforce that every node of \( G \) has at most one outgoing edge and at most one incoming edge. It is also easy to enforce no self loop in \( G \). By negating the following patterns we also make sure that the next element of a \( N_{\text{last}} \) node can only be a \( N_{\text{beg}} \) node.

\[
\begin{array}{c}
N_{\text{last}} \quad R_{\text{inner}} \\
\frac{1}{2} \quad \frac{1}{2}
\end{array}
\]

It remains to take care of loops and of sequences that may stop abruptly. For this we use function calls and compute, step by step, the transitive closure of \( G \) and the nodes at distance \( 2^{2^{2n+1}} \). Then suitable Tree-LTL formulas can check that \( G \) has the right format.

The transitive closure of \( G \) is computed by induction as follows: If \( T \) is the relation computed at some step, the next step computes \( \exists z T(x, z) \land T(z, y) \). With a recursion-free GXML of size \( n \), runs can only have about \( 2^n \) steps. Hence we can’t compute the complete transitive closure of \( G \) but only the first \( 2^{2n+1} \) steps. But with the divide and conquer algorithm described above, this is enough to detect the
loops of $G$ of length smaller than $2^{2n+1}$. This is enough for simulating $M$, because $M$ on $w$ ends in fewer than $2^{2n}$ steps!

We now specify the function calls that compute the transitive closure of $G$ as described above. To do this we use $2n+5$ functions, $f, f_0, \ldots, f_{n+1}, f'_0, \ldots, f'_{n+1}$. We add below the root $M$ a function $f$ which is always enabled (its call guard is always true) and whose argument query computes $G$:

![Diagram](image)

Each $f_i$, for $i \leq n$, is defined so that it is replaced by $!f_{i+1}f'_{i+1}$ or by $!f_{n+1}$. This is done by making its call guard and return guard always true, its argument query void, and its return query the forest $!f_{i+1}f'_{i+1}$, if $i < n$, or the tree $!f_{n+1}$, if $i = n$.

Each $f'_i$ is defined similarly except for the call guard which is enabled only if no node containing a call to some $f_j$ is present in the tree. It is easy to see, by induction on $i$, that each $f_i$ and each $f'_i$ is eventually executed $2^n$ times.

The function $f_{n+1}$ performs one step of the computation of the transitive closure of $G$. We have seen that it will eventually be executed $2^{n+1}$ times as desired. The call guard of $f_{n+1}$ is always true. The input query of $f_{n+1}$ does the expected computation and is given by:

![Diagram](image)

The return guard of $f_{n+1}$ is always true and returns the answer of the input query.

By negating the following pattern in $S$ we enforce that a self loop will never occur and hence that $G$ has no small loops:

![Diagram](image)

The following Tree-LTL formula now checks that $G$ induces a long path (i.e. of length $2^{2n+1}$) from the $N_{\text{beg}}$-node of the $R_{\text{beg}}$-subtree to the $N_{\text{last}}$-node of the $R_{\text{last}}$-subtree. It does so by checking that the corresponding pair of data values eventually occur in the computation of the transitive closure but only in the very last step. This last step can be detected by a tree pattern (denoted by $\text{LAST}$ below) that checks that the evaluation of the transitive closure is over: no more running calls besides $f$. We denote by $\text{UNTIL}$ the variant of $\varphi \cup \psi$ that requires $\psi$ to eventually hold.

![Diagram](image)

It now remains to check that the distance in $G$ between a $N_{\text{beg}}$-node and a $N_{\text{last}}$-node of a same $R$-subtree is exactly $2^n$. We do this by checking that (i) it is the case for the $R_{\text{beg}}$-subtree (ii) all $R$-subtrees have the same distance between their $N_{\text{beg}}$-node and a $N_{\text{last}}$-node.

Condition (i) is checked by a Tree-LTL formula as above except that the corresponding pair of data value must now occur at the middle step (step number $2^n$ out of $2^{n+1}$) of the computation of the transitive closure and not before. Note that the middle step can be detected via a tree pattern (denoted $\text{MIDDLE}$ below) because it is exactly when $f'_0$ is called.

![Diagram](image)

We now turn to condition (ii). We set the DTD so that each $U$-node also has a data value attached to it (not depicted in the figure of a configuration of $M$ above). Similarly as above we use $U_{\text{beg}}$ and $U_{\text{last}}$ as shorthand for $U$-nodes whose parent is a $N_{\text{beg}}$-node or $N_{\text{last}}$-node. We use these data values as an index of the position (relative to the order induced by $G$) of the corresponding $U$-node within the $R$-subtree. We first make sure that within the same $R$-subtree, no two $U$-nodes have the same data value. This is done by forbidding the pattern:

![Diagram](image)

$z \neq z'$
Hence a data value uniquely determines a position within an \( R \)-subtree. We now make sure that all successive \( R \)-subtrees have the same sequence of data values under the \( U \)-nodes. This is done as follows. For \( U_{\text{beg}} \) and \( U_{\text{last}} \) nodes a pattern can easily enforce this by stating that two \( U_{\text{beg}} \)-nodes (and two \( U_{\text{last}} \)-nodes) cannot carry different data values. This property is then propagated via the successor relation by forbidding the pattern depicted below:

$$
\begin{array}{c}
\text{N} & \text{P} & \text{M} & \text{N} & \text{P} \\
\text{y} & \text{z} & \text{y} & \text{z}' & \\
\text{x} & \text{x} & \text{x} & \text{x} & \text{x}
\end{array}
$$

Notice in the pattern above how the variable \( x \) is used to make sure that \( y \) and \( z \) are the data values associated to two successive positions (similarly for \( x' \) with \( y' \) and \( z' \)).

Notice that the above constraints imply that all successive \( R \)-subtrees must have the same length. Since the initial \( R \)-subtree has length \( 2^n \), this implies that all successive \( R \)-trees also have the desired length \( 2^n \). This concludes the description of the successor relation. Let us denote by \( \phi_{NP} \) the conjunction of all the Tree-LTL formulas mentioned above. It follows from the discussion above that if there is a run from the initial instance that satisfies \( \phi_{NP} \), then the initial instance is such that \( G \) codes an appropriate successor relation.

**Simulation of \( M \)** It is now easy to check that the initial configuration, the \( R_{\text{beg}} \)-subtree, is indeed initial. This can be done with patterns that check the coding of \( w \) using the successor relation induced by \( G \).

Consistency of the transitions of \( M \) can now be enforced by forbidding patterns of the form (this is a classical construction that we do not detail here):

$$
\begin{array}{c}
\text{N} & \text{P} & \text{M} & \text{N} & \text{P} \\
\text{y} & \text{z} & \text{y} & \text{z} & \\
\text{x} & \text{x} & \text{x} & \text{x} & \text{x}
\end{array}
$$

Notice in the pattern above how the variable \( x \) is used to make sure that the two \( R \)-subtrees correspond to two successive configurations (\( N_{\text{last}} \) denotes the marked node for the end of the sequence inside an \( R \)-subtree while \( P_{\text{last}} \) denotes the beginning). Similarly, \( y \) (resp. \( y' \)) is used to make sure that \( a \) and \( b \) (resp. \( c \) and \( d \)) are consecutive positions in their respective configuration. The variable \( z \) enforces (this is where we use the fact that a data value determines a position) that the position of \( a \) is the same as the position of \( c \). It is then a straightforward procedure to allow only the right combination of patterns with the appropriate \( a,b,c \) and \( d \) in order to simulate the transitions of \( M \).

It now remains to check that the accepting configuration is reachable from the initial one. Whether a configuration is accepting can easily be checked using a pattern. The configuration reached in \( 2^{2^k} \) steps from the initial one can be extracted from the computation of the transitive closure of the successor relationship among the \( R \)-subtrees: it is available under the subtree rooted by \( a_{\ell} \). Hence a Tree-LTL formula can check that this last configuration is accepting. Let us denote this formula by \( \phi_{\text{accept}} \).

**Wrapping up.** The recursion-free GAXML schema \( S \) is defined as:

- \( \Delta \) is the DTD given above for the global shape of the initial instance together with all the patterns introduced above that guarantee that the transitions are correct.
- The functions are those described above. They enforce the right size of the initial instance.

Altogether the construction of \( S \) is polynomial in \( M \) and \( w \).

The Tree-LTL formula \( \varphi \) is \( \neg (\phi_{NP} \land \phi_{\text{accept}}) \). It is also constructed in polynomial time from \( M \) and \( w \). Note that the patterns in the formula do not contain any free variables.

It is now easy to check that \( M \) rejects \( w \) iff all runs of \( S \) verify \( \varphi \). Once the initial instance is provided either it does not code a valid run of \( M \) and then \( \phi_{NP} \) does not hold, or it does code a valid run of \( M \), but then \( \neg \phi_{\text{accept}} \) guarantees that the run is not accepting. Hence \( M \) cannot accept \( w \). □

### 6.5 Undecidability results

In this and other proofs we use a reduction from the implication problem for functional and inclusion dependencies, known to be undecidable. We briefly recall this problem (see [3] for more details). Let \( R \) be
a relation. An inclusion dependency (ID) over \( R \) is an expression \([A] \subseteq [B]\) where \( A \) and \( B \) are sets of attributes of \( R \) of the same size. \( R \) satisfies \([A] \subseteq [B]\) if \( \pi_A(R) \subseteq \pi_B(R) \). A functional dependency (FD) over \( R \) is an expression \( V \rightarrow C \), where \( V \) is a set of attributes and \( C \) an attribute of \( R \). Relation \( R \) satisfies \( V \rightarrow C \) if no two tuples of \( R \) agree on \( V \) and disagree on \( C \). The implication problem asks, given a set \( \Gamma \) of IDs and FDS, and an FD \( F \) over \( R \), whether \( \Gamma \models F \), i.e. every finite \( R \) that satisfies \( \Gamma \) must also satisfy \( F \).

**Theorem 4.1** It is undecidable, given a positive tree pattern \( P \) without variables and a GAXML schema \( S \) with no data constraints or external functions, satisfying the non-restrictiveness conditions relaxed by any of (1), (2), or (3) above, whether some instance satisfying \( P \) is reachable in a run of \( S \).

**Proof:** We use a reduction from the implication problem for FDS and IDs. Let \( R \) be a relation with \( k \) attributes, \( \Gamma \) a set of FDS and IDs over \( R \), and \( F \) an FD over \( R \). For (1-3) we construct a GAXML schema \( S \) satisfying the stated restrictions, and a tree pattern \( P \), such that \( \Gamma \nmid F \) iff some instance satisfying \( P \) is reachable in a run of \( S \). We represent relation \( R \) with attributes \( A_1 \ldots A_k \) in the standard way, as a tree described by the DTD\(^5\)

\[
R \rightarrow T^* \\
T \rightarrow A_1 \ldots A_k \\
A_i \rightarrow \text{dom}
\]

Consider (1). Suppose the DTD of \( S \) allows an unbounded number of function calls in valid trees. In order to check the inclusion dependencies, we use one internal function \( f_r \) for each ID \( \tau \in \Gamma \), and one additional internal function \( g \). Their call guards will be described shortly. Their argument queries are largely irrelevant – we assume they are trivial and produce the empty forest. Their return guards are similarly defined as \( \text{false} \), so no answer is ever returned. We add one node labeled \( !f_r \) under each node \( T \), for each ID \( \tau \in \Gamma \). Finally, we add one node labeled \( !g \) under \( R \). Thus, an initial instance of \( S \) is of the form:

\[
\begin{array}{c}
A_1 \\

A_2 \\

\vdots \\

A_k \\

!f_r \\

!g \\

R
\end{array}
\]

The guard of each \( f_r \) checks that the inclusion dependency \( \tau \) is not violated for the tuple local to the node labeled \( !f_r \). For example, if \( \tau = R[A_i] \subseteq R[A_j] \), the guard of \( f_r \) is

\[
\begin{array}{c}
A_1 \\

A_2 \\

\vdots \\

A_k \\

\vdots \\

\downarrow ^{\text{self}} \\

\downarrow ^{!f_r}
\end{array}
\]

The guard \( \gamma(g) \) of \( g \) is the conjunction of several tree patterns. The first simply checks that no node labeled \( !f_r \) exists in the tree. This ensures that all calls to the \( !f_r \) have been made, which implies that their guards were true, so no \( \tau \in \Gamma \) is violated. Satisfaction of the FDSs in \( \Gamma \) is ensured by adding to \( \gamma(g) \) the obvious negative patterns forbidding violations. Finally, violation of \( F \) is ensured by a positive pattern, also added to \( \gamma(g) \). The pattern \( P \) simply checks that a node labeled \( ?g \) exists in the tree, so the guard of \( g \) is true. Clearly, \( P \) is reached in a run of \( S \) iff there exists \( R \) that satisfies \( \Gamma \) and violates \( F \), iff \( \Gamma \nmid F \).

Next, consider (2) and (3). We only outline the proof for (2), since (3) is very similar. Thus, \( S \) is allowed to use continuous functions (but all other restrictions remain in force). As above, suppose \( \Gamma \) and \( F \) apply to a relation \( R \) with attributes \( A_1 \ldots A_k \). The idea of the proof is as follows. As above, the FDSs can be easily checked using tree patterns. In order to check satisfaction of the IDs, we augment \( R \) with two attributes \( B, C \) meant to represent a successor (or almost) on the tuples of \( R \), yielding \( \tilde{R} \). The IDs are checked by stepping through the tuples one-by-one using the successor relation, and verifying for each that no ID is violated. This is done using continuous functions. We next provide more details.

The relation \( \tilde{R} \) is represented as above. To this we add continuous functions \( g, h \) and \( c \), with respective parents \( G, H \) and \( E \), all under root \( \tilde{R} \). For technical reasons we need one additional function \( \text{init} \) that appears under \( G \) and \( H \). The DTD rules for \( G, H \) and \( E \) are:

\[
G \rightarrow (\langle g + ?g \rangle(\text{init dom} + ?\text{init dom})) \\
H \rightarrow (\langle h + ?h \rangle(\text{init dom} + ?\text{init dom})) \\
E \rightarrow \langle e + ?e \rangle
\]

The role of \( \text{init} \) is simply to enforce the presence of a single data value under \( G \) and \( H \) in the initial instance. Multiple data values may appear once \( \text{init} \) has been called. We make the return guard of \( \text{init} \) \( \text{false} \), so that no answer is ever returned.

Thus, an initial instance of \( S \) has the following shape:

---

\(^5\)This classical notation maps in the obvious way to constraints in our DTDs.
We denote by $G_{BC}$ the graph whose nodes are the data values occurring under the $B$’s, and for which there is an edge from $\gamma$ to $\delta$ iff $\gamma$ occurs under $B$ and $\delta$ under $C$ in the same tuple. We also use two constant data values start and end. The call guard of $!g$ ensures the following:

(i) $B$ and $C$ are keys for relation $\bar{R}$ and their values are distinct in every tuple;

(ii) start occurs under some $B$ node and has in-degree zero in $G_{BC}$;

(iii) start occurs under $G$;

(iv) end does not occur under $G$.

Note that (i) ensures that each tuple in $R$ is uniquely identified by the data value under the $B$ attribute (we say that each tuple is indexed by the corresponding $B$ value). Also, (i) ensures that, in the graph $G_{BC}$, all nodes have in-degree and out-degree at most one. The role of function $g$ is to compute all data values that are reachable from start in $G_{BC}$. This can be easily done by repeatedly calling $!g$ and producing, after $m$ calls, all $\gamma$ reachable from start in $G_{AB}$, for which the distance from start to $\gamma$ is at most $m$. Specifically, the argument query of $g$ has head $\{Y\}$ and body

$$\begin{align*}
\bar{R} & \leftarrow T \leftarrow Y \leftarrow G \\
B \leftarrow C & \leftarrow R \leftarrow X \\
X & \leftarrow Y
\end{align*}$$

Its return guard is true and the return query returns everything under $a_g$ (so the result of the argument query is simply returned without change). Note that (iv) forces $!g$ to stop as soon as end is reached (if at all).

The guard of $h$ checks the following:

- end occurs under $G$ (so $g$ has been called at least once and end is reachable from start). Note that if this holds, there is a unique simple path from start to end in $G_{BC}$, and the data values under $G$ are exactly the nodes along the path;

- start occurs under $H$ (and therefore is the only data value under $H$ in the initial instance);

- end does not occur under $H$.

The role of $h$ is to redo the computation performed by $!g$, with the additional task of checking, for each data value $\gamma$ reached from start, that the tuple of $R$ indexed by $\gamma$ does not violate any ID of $\Gamma$ within the sub-relation of $R$ consisting of the tuples indexed by data values along the path from start to end (available under $G$), which we denote by $R_{sc}$. This is done by including appropriate sub-patterns in the body of the argument query of $h$. The portion of the body detecting a new reachable value $Y$ is the same as for $g$, except that $G$ is replaced by $H$. This is augmented in order to check each ID $\tau$ in $\Gamma$. For example, if $\tau$ is an ID $R[A_{ij}] \subseteq R[A_{ij}]$, the body of the argument query of $!h$ is augmented as follows:

Finally, we turn to $e$. The return guard of $e$ checks the following:

- end occurs under $H$;
- all FDs in $\Gamma$ are satisfied by $R_{sc}$;
- the FD $F$ is violated by $R_{sc}$.

Thus, the guard of $e$ becomes true iff $R_{sc}$ satisfies $\Gamma$ and violates $F$. The pattern $P$ simply checks that $?e$ occurs in the tree. Clearly, an instance satisfying $P$ is reachable in a run of $S$ if $\Gamma \not\models F$.

**Theorem 4.2** It is undecidable, given a positive pattern $P$ without variables and a GAXML system $S$ with no data values and no external functions, satisfying the non- recursiveness conditions relaxed by allowing a cyclic call graph, whether some instance satisfying $P$ is reachable in a valid run of $S$.

**Proof (sketch)** The proof is by reduction from reachability for deterministic two-counters machines. These are finite state automata with two counters and an initial state. Each transition depends on the current state and whether the counters have value zero. A transition may change the current state and increment or decrement one of the counters. It is known
that it is undecidable whether a given state is reachable for deterministic counter machines [19].

Let $M$ be a deterministic counter machine with states $\{q_0, \ldots, q_n\}$, initial state $q_0$, and counters denoted $C_1, C_2$. We simulate a run of $M$ using a GAXML system $S$ as follows. For each state $q_i$ of $M$ we assume a function, also denoted $q_i$ by slight abuse. (recall that all functions of $S$ are non-continuous). We also assume two functions $c_1, c_2$ used to code the two counters. The value of counter $C_i$ is the number of active running calls to $c_i$ in the instance. To facilitate the simulation of transitions, we represent both the current and next state of $M$, under distinct roots Current and Next. The DTD enforces that at most one function call occurs under each of Current or Next. We assume wlog that the start state $q_0$ of $M$ is never used again in the course of the computation. The current state $q$ of $M$ is indicated by the presence of a call $?q$ as a child of Current. The start configuration of $M$ (current start state $q_0, C_1 = C_2 = 0$) is represented as follows:

![Diagram](attachment:diagram.png)

Each call to $c_i$ produces a new running call containing a single call $!c_i$ (in addition to some other functions used for bookkeeping and described below). Thus, the number of running calls to $c_i$ is simply the number of nodes labeled $?c_i$. The last running call is the only one containing $!c_i$. Incrementing the counter $C_i$ in $M$ is simulated by making a new call to $c_i$. Decreasing $C_i$ is done by returning the answer to the last call to $c_i$. A transition of $M$ from state $q_i$ to $q_j$ is simulated in several steps:

- Call function $q_j$ under Next (its guard tests the condition of the transition: the current state is detected by finding $?q_j$ under Current, and a zero counter $C_i$ is detected by the absence of $?c_i$).
- The counters are incremented or decremented by making a new call or returning the result to the last call of the corresponding function.
- The call $?q_i$ under Current returns $!q_i$ and a new call to $q_i$ (for which $?q_i$ appears under Next) is made.
- The call $?q_j$ under Next returns $!q_j$.

Implementing the above sequence requires some careful control achieved by a few additional functions. We omit the details. Finally, let $P$ be the positive pattern checking the presence of $?q$. It is clear that $M$ reaches state $q$ iff an instance satisfying $P$ can be reached in a run of $M$. 

\[ \square \]

**Theorem 4.4** It is undecidable, given a positive tree pattern $P$ without variables, and a recursion-free GAXML schema $S$ with no data constraints and no external functions, using tree patterns with negative sub-patterns, whether there exists an instance satisfying $P$ that is reachable in a run of $S$.

**Proof:** We use again a reduction from the implication problem for FDs and IDs. Let $\Gamma$ be a set of FDs and IDs and $F$ an FD over a relation $R$. We build $P$ and recursion-free $S$ such that $\Gamma \not\models F$ iff some instance satisfying $P$ is reachable in a run of $S$. The key observation is that one can easily check for violation of an ID using a pattern with a negative sub-pattern (so its negation states satisfaction of the ID). Satisfaction of the FDs in $\Gamma$ and violation of $F$ are tested as before. All the conditions can be placed in the guard of a function $g$. The pattern $P$ simply tests the existence of a call $?g$.

\[ \square \]

**Theorem 4.5** It is undecidable, given a recursion-free GAXML schema $S$ and an $\exists$Tree-LTL sentence $\varphi$, whether $S$ satisfies $\varphi$.

**Proof:** We use a reduction from the implication problem for FDs and IDs. Let $\Gamma$ be a set of FDs and IDs, and $F$ an FD over a relation $R$. We exhibit a recursion-free GAXML schema $S$ and an $\exists$Tree-LTL sentence $\varphi = \exists X (\neg \xi(X))$ such that $S \models \varphi$ iff $\Gamma \not\models F$. Equivalently, we show that there exists a run of $S$ satisfying $\forall X \xi(X)$ iff $\Gamma \not\models F$. We represent relation $R$ as above, and use one (internal or external) function $g$ whose guard enforces satisfaction of the FDs in $\Gamma$ and violation of $F$. The formula $\xi$ contains several conjuncts. For each ID $\tau = [A] \subseteq [B]$ of $\Gamma$, $\xi$ contains a conjunct stating that, if $X_\tau$ is a tuple in $R$, then $\pi_A(X_\tau) \subseteq \pi_B(X_\tau)$. This can be expressed by tree patterns with free variables $X_\tau$. Finally, $\xi$ includes the conjunct $X (//?g)$ stating that $g$ is called in the first transition. Let $X$ consist of all the variables occurring in $X_\tau$, for $\tau \in \Gamma$. Clearly, $\forall X \xi(X)$ is satisfied in a run of $S$ iff the relation $R$ represented in the initial instance satisfies $\Gamma$ and violates $F$, i.e. $\Gamma \not\models F$. 

\[ \square \]
Theorem 4.6 It is undecidable, given a positive tree pattern \( P \) without variables and a recursion-free GAXML schema \( S \), if \( S \) satisfies\(^6\) \textbf{AXEG} (\( \neg P \)).

Proof: We prove, equivalently, that it is undecidable whether there exists an initial instance of \( S \) satisfying \( \textbf{AFP} \). We use, once more, a reduction from the implication problem for FDs and IDs. Let \( \Gamma \) be a set of FDs and IDs and \( F \) an FD over a relation \( R \) with attributes \( A_1, \ldots, A_k \). We outline the construction of a recursion-free GAXML schema \( S \) and a pattern \( P \) such that \( \Gamma \not\models F \) if there exists an initial instance \( I_0 \) of \( S \) for which \( P \) is reachable in all runs starting at \( I_0 \). We represent \( R \) in the standard way, and use two functions \( f \) and \( g \). The function \( f \) is external and returns a single tuple in \( R \) (this can be enforced by the DTD and a data constraint). The function \( g \) is internal. Its guard checks that the FDs in \( \Gamma \) are satisfied, \( F \) is violated, and the tuple of \( R \) returned by \( f \) satisfies all the IDs of \( \Gamma \). The pattern \( P \) checks for the occurrence of \( ?g \). Clearly, \( ?g \) is reached from \( I_0 \) in all runs iff the relation \( R \) represented by \( I_0 \) satisfies \( \Gamma \) and violates \( F \). \( \square \)

6.6 Decidability in the general setting

Theorem 4.7 (Safety) It is decidable in \textsc{co-NP-time}, given a GAXML schema \( S \) and a Boolean combination \( \varphi \) of tree patterns, whether (i) all initial instances of \( S \) satisfy \( \varphi \), and (ii) for all valid instances \( I \) and \( J \) of \( S \) such that \( I \vdash J \), if \( I \models \varphi \) then \( J \models \varphi \).

Proof: The proof uses Lemmas 6.1 and 6.2, which recall do not assume non-recursive schemas. Let \( S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta) \) be a GAXML schema. Consider (i). Let \( \Delta_0 \) be the DTD of \( \Delta \) and \( \psi \) its data constraint (a Boolean combination of tree patterns). We need to show that it is decidable whether (i) there exists a tree with no nodes labeled \( ?f \), satisfying \( \Delta_0 \) and \( \psi \land \neg \varphi \). We can easily modify \( \Delta_0 \) in linear time so that \( ?f \) is disallowed. So, suppose no valid tree contains \( ?f \). We show that if (i) holds, then there exists a tree \( I_0 \) with the same property and of size exponential in \( |\Delta| + |\varphi| \). Indeed, suppose \( I \) satisfies (i). Let \( \mathcal{P} \) be the set of tree patterns occurring in \( \psi \) or \( \varphi \) that hold in \( I \), and let \( \mathcal{M} \) consist of one matching into \( I \) for each \( P \in \mathcal{P} \). Let \( I_0 \) be the minimal prefix of \( I \) containing all nodes in the images of matchings in \( \mathcal{M} \). Note that \( |I_0| \leq d \cdot (|\psi| + |\varphi|) \), where \( d \) is the maximum depth of a tree satisfying \( \Delta_0 \). Finally,

let \( I_0 \) be the completion of \( I_0 \) wrt \( \Delta_0 \). By Lemma 6.2, \( |I_0| \leq d \cdot (a \cdot \max(\Delta))^d \cdot |I_0| \), where \( a \) is the size of the alphabet of \( \Delta_0 \) and \( \max(\Delta) \) is the maximum integer used in the specification of \( \Delta_0 \). Thus, \( |I_0| \leq d^2 \cdot (a \cdot \max(\Delta))^d \cdot (|\psi| + |\varphi|) \), and \( I_0 \) is exponential in \( |\Delta| + |\varphi| \). Clearly, \( I_0 \) satisfies \( \Delta_0 \) and \( \psi \land \neg \varphi \).

Now consider (ii). Once again, we use a small model property. Suppose there exist valid instances \( I \) and \( J \) of \( S \) such that \( I \models J \), if \( I \models \varphi \) but \( J \models \varphi \). We can show that there exist valid instances \( I_0 \) and \( J_0 \) of \( S \), of size exponential in \( |\Delta| + |\varphi| \), such that \( I_0 \models \varphi \) but \( J_0 \models \varphi \). The proof is essentially a special case of the proof of Lemma 3.4, for the case of runs of length 2. We omit the straightforward details. \( \square \)