# Static Analysis of Active XML Systems

**Serge Abiteboul*** INRIA-Saclay & U. Paris Sud, France

**Luc Segoufin** INRIA & LSV - ENS Cachan, France

**Victor Vianu†** U.C. San Diego, USA

**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 [25] with embedded function calls realized as Web service calls [26]. In the spirit of [20, 23], 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. An Active XML system specifies a set of interacting AXML documents. 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 sub-tasks.

*Work supported by the Agence Nationale de la Recherche under grant Docflow, 06-MDCA-005.

†Work supported in part by the National Science Foundation under grant number IIS-0415257.

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 restriction 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 *Guarded AXML*, that we briefly describe next.

In Guarded 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 types 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 how our results can be 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 function calls in runs. We prove that for such recursion-free GAXML, the satisfaction of Tree-LTL formulas is CO-2NEXPTIME-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) deal with documents that do not evolve in time. Typically, they consider the consistency problem for XML specifications using DTDs and (foreign) key constraints [7, 8], the query containment problem [5] or the type checking problem [9]. This motivated studies of automata and logics on strings and trees over infinite alphabets [22, 13, 10]. See [24] 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 non-monotone systems.

Verification of temporal properties of Web services has mostly been considered using models abstracting away data values (see [19] for a survey). Verification of data-intensive Web services is studied in [14, 16], and a verifier implemented [15]. 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 incomparable.

The related work that is perhaps closest to this paper is [18], where a tree pattern rewriting systems is introduced to model the evolution of dynamic XML documents. The rewriting system may be recursive, but XML documents have no data values in their model. The main result is that reachability of a tree satisfying a specified pattern is decidable under certain conditions. This is orthogonal to our results, because of the absence of data values.

**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. Finally, extensions of our model and decidability results to compositions of GAXML systems are presented in Section 5. The paper concludes with a brief discussion.

# 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 revisit this issue in Section 5). To illustrate our definitions, we use fragments of a Mail Order GAXML processing system, detailed 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* $\mathcal{N}$ (denoted $x, y$), *tags* $\Sigma$ (denoted $a, b, c, \ldots$), *function symbols* $\mathcal{F}$, *data values* $\mathcal{D}$ (denoted $\alpha, \beta, \ldots$), *data variables* $\mathcal{V}$ (denoted $X, Y, Z, \ldots$), possibly with subscripts. We also use two sets of *marked function symbols*, $\mathcal{F}^! = \{!f \mid f \in \mathcal{F}\}$ and $\mathcal{F}^? = \{?f \mid f \in \mathcal{F}\}$. 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, but whose result has not yet been returned (running call). We denote by $\mathcal{F}^{?!}$ the union $\mathcal{F}^! \cup \mathcal{F}^?$.

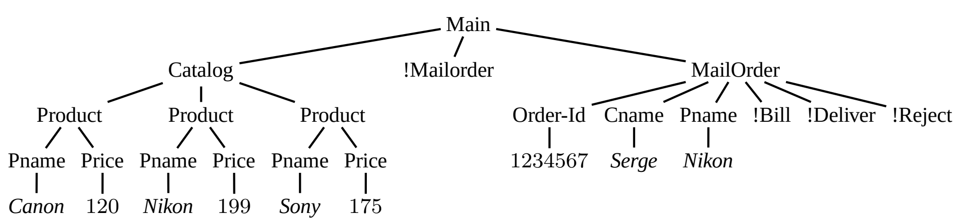



Figure 1: A GAXML document.

A *Guarded AXML* (GAXML) document is a tree whose internal nodes are labeled with tags in $\Sigma$ and whose leaves are labeled by either tags, marked function symbols, or data values. A GAXML forest is a set of GAXML trees. An example of GAXML document is given in Figure 1 (see Appendix for the full specification of the Mail Order example).

To avoid repetitions of isomorphic sibling subtrees, we define the notion of reduced tree. Two trees $T_1$ and $T_2$ are *isomorphic* iff 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 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 controlling the activation of function calls and as a basis for our query language. Patterns are constructed from tree patterns that we define first. 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. Formally, a *tree pattern* $p$ is a tuple $(M, G, \lambda_M, \lambda_G)$, where:

- $(M, G)$ is a finite tree with $M \subset \mathcal{N}$,
- $\lambda_M : M \to \Sigma \cup \mathcal{F}^{?!} \cup \mathcal{D} \cup \mathcal{V} \cup \{*\}$ is a node labeling function such that $\lambda_M(x) \in \Sigma \cup \{*\}$ for every internal node $x$,
- $\lambda_G : G \to \{/, //\}$.

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 labels in $\Sigma \cup \mathcal{F}^{?!} \cup \mathcal{D}$, (iv) nodes in $p$ labeled with variables are mapped to nodes in $T$ labeled with data values, and (v) if two nodes $x, x'$ are labeled with the same variable $X$, the nodes $\mu(x), \mu(x')$ must be labeled with the same data value.

If $\mu$ maps a node $x$ labeled with some variable $X$ to some node labeled with some data value $\alpha$, we say by extension that $\mu(X) = \alpha$. Note that this is well defined because of (v). Observe that nodes labeled with $*$ are unrestricted, so $*$ acts as a wildcard.

A *pattern* $P$ is a pair $(\{p_1, \ldots, p_n\}, \mathit{cond})$, 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 $X = Y$. A matching of $P = (\{p_1, \ldots, p_n\}, \mathit{cond})$ 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. More precisely, if $X = \alpha$ is in *cond*, then $\mu(X) = \alpha$ if $\alpha \in \mathcal{D}$ and $\mu(X) = \mu(\alpha)$ if $\alpha \in \mathcal{V}$. And similarly, for $X \neq \alpha$.

An example is given in Figure 2 (a). The pattern shown there expresses the fact that the value `Order-Id` is not a key. It does not hold on the GAXML document of Figure 1. (Indeed, we want `Order-Id` to be a key.) We say that a pattern $P$ holds in a forest $F$ iff there exists at least one matching of $P$ into $F$. We then say that $P(F)$ is true, otherwise it is false. This definition extends to Boolean combination of patterns by replacing each pattern $P$ by $P(F)$. In particular this means that the patterns are matched independently of each other: If a variable $X$ occurs in two different patterns $P$ and $P'$ of the Boolean combination, then it is treated as quantified existentially in $P$ and independently quantified in $P'$.

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 ($P$, *self*) where $P$ is a pattern and *self* is a node of $P$. A relative pattern ($P$, *self*) is evaluated on a pair $(F, x)$ where $F$ is a forest and $x$ is a node of $F$. Such a pattern forces the node *self* in the pattern to be mapped to $x$. 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 `!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.

**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

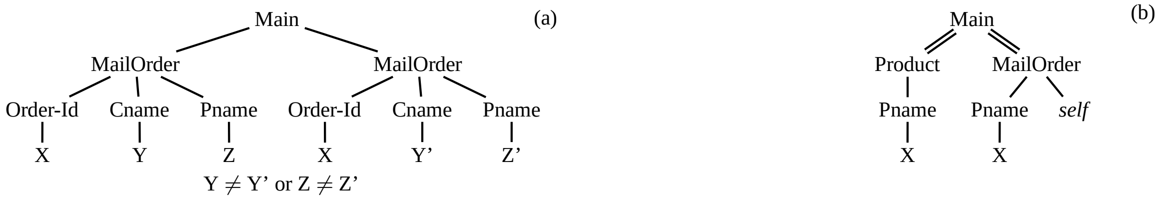



Figure 2: Two patterns

valuations of its 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 *Body* $\rightarrow$ *Head*, where *Body* is a pattern and *Head* is a forest such that, for each tree $H$ of *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`.

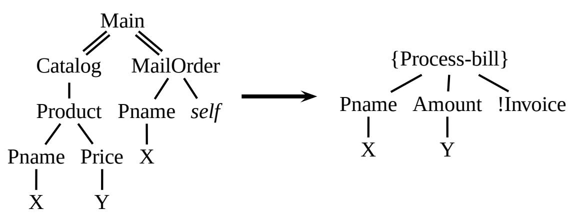



Figure 3: Example of a relative query

Let $F$ be a forest and $Q$ = *Body* $\rightarrow$ *H* a query with a single tree for head. Let $\mathcal{M}$ 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 $\mu \in \mathcal{M}$, 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 $\{\mu(H_c) \mid \mu \in \mathcal{M}\}$. Note that if $\mathcal{M} = \emptyset$ 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* $\rightarrow H_1, ..., H_n$. Then $Q(F) = \cup Q_i(F)$ where for each $i$, $Q_i$ = *Body* $\rightarrow$ $H_i$.

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

**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 relative query of Figure 3 on the GAXML document of Figure 1 at the unique node labeled `!Bill`. There is a unique matching of the *Body* pattern and the result is isomorphic to the *Head* tree of the query with $X$ replaced by *Nikon* and $Y$ by 199 (with no parenthesis for `Process-bill`).

**DTD** Trees used by a GAXML system may be constrained using DTDs and tree pattern formulas. For DTDs, we use a typing mechanism that restricts, for each tag $a$, the labels of children that $a$-nodes may have. As our trees are unordered, the DTD constrains, for each node, the number of children with given labels. More precisely, a DTD is a triple $(\Sigma_0, R, \mathcal{R})$, where $\Sigma_0$ is a finite subset of $\Sigma$, $R \subseteq \Sigma_0$ is the set of allowed root labels, and $\mathcal{R}$ is a finite set of rules $a \to \psi$ where $a \in \Sigma_0$ and $\psi$ is a Boolean combination of inequalities of the form $|b| \geq k$ where $b \in \Sigma_0 \cup \mathcal{F}^{?!} \cup \{\mathit{dom}\}$ and and $k$ is a non-negative integer[1] (here, *dom* is a symbol that stands for any data value). A tree $T$ satisfies a DTD $(\Sigma_0, r, \mathcal{R})$ if all its tags are in $\Sigma_0$, its root has label in $R$, and for each rule $a \to \psi$, and each node labeled $a$, its children satisfy the condition $\psi$ (details are omitted). If there is no rule in $\mathcal{R}$ for some $a \in \Sigma_0$, then all nodes labeled $a$ must be leaves. A forest $F$ satisfies a DTD if each tree in $F$ satisfies it.

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

- $\Phi_{\text{int}}$ is a finite set of internal function specifications.
- $\Phi_{\text{ext}}$ is 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 (called *data constraint*).

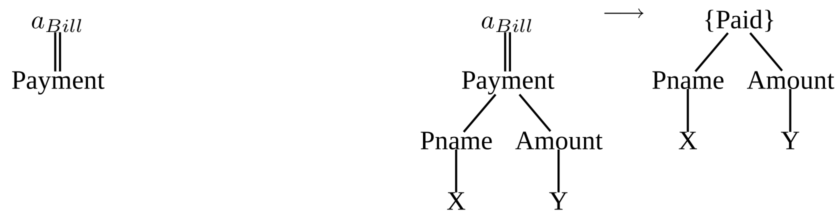
We next detail $\Phi_{\text{int}}$ and $\Phi_{\text{ext}}$. For each $f \in \mathcal{F}$, let $a_f$ be a new distinct label in $\Sigma$. Intuitively, $a_f$ will be the label of the root of a tree where a call to $f$ will be evaluated. (This tree may be seen as work space for the evaluation of the function.)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

- $f \in \mathcal{F}$ is the name of the function.
- $\mathit{arg}(f)$ (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.
- $\mathit{kind}(f) \in \{$*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.
- $\mathit{ret}(f)$ (the *return query*) is a query rooted at $a_f$.

By slight abuse, if $\Phi_{\text{int}}$ contains the specification of a function with name $f$, we say that $f$ is in $\Phi_{\text{int}}$.

[1]For the purpose of complexity analysis, we take the size of $|b| \geq k$ to be $k$. This is commensurate with the classical specification of DTDs using regular expressions.

**Example 2.2** We continue with our running example. The function `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 `Invoice` is an external function eventually returning `Payment` (with product and amount paid), the return guard and return query of `Bill` are:




*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 *ret*$(f)$ are missing. Intuitively, an external call can return any answer at any time. Its answer is however constrained by $\Delta$.

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

1. For each $x$ with label $?f$, *eval*$(x)$ is a tree in $\mathcal{T}$ with root label $a_f$.

2. Every tree in $\mathcal{T}$ with root label $a_f$ is *eval*$(x)$ for some $x$ labeled $?f$.

An instance $(\mathcal{T}, \mathit{eval})$ of $S$ is *valid* if $\mathcal{T}$ satisfies $\Delta$.

**Runs** Let $I = (\mathcal{T}, \mathit{eval})$ and $I' = (\mathcal{T}', \mathit{eval}')$ 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 $x$ in $T \in \mathcal{T}$, labeled $!f$ for $f \in \Phi_{\text{ext}}$, such that $\gamma(f)(\mathcal{T}, x)$ holds, where $\gamma(f)$ is the call guard of $f$; and $I'$ is obtained from $I$ by changing the label of $x$ to $?f$.

*Internal call:* there exists some node $x$ in $T \in \mathcal{T}$, labeled $!f$ for $f \in \Phi_{\text{int}}$, such that $\gamma(f)(\mathcal{T}, x)$ holds, where $\gamma(f)$ is the call guard of $f$; and $I'$ is obtained from $I$ by changing the label of $x$ to $?f$ and adding to the graph of *eval* the pair $(x, T')$, where $T'$ is a tree consisting of a root $a_f$ connected to the forest that is the result of evaluating the argument query *arg*$(f)$ on input $(\mathcal{T}, x)$. (All nodes occurring in $T'$ are new.)

*Return of internal call:* there is some node $x$ labeled $?f$ in some tree of $\mathcal{T}$, where $f \in \Phi_{\text{int}}$, such that $T = \mathit{eval}(x)$ 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 *ret*$(f)$ on $T$ and add the resulting forest as a sibling of the node $x$;

- remove *eval*$(x)$ from $\mathcal{T}$ and $x$ from the domain of *eval*;

- if $f$ is non-continuous remove the node $x$, otherwise change $x$'s label to $!f$.

*Return of external call:* there exists some node $x$ labeled $?f$ in some tree of $\mathcal{T}$, 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 *eval* and (ii) the result (a forest with labels in $\Sigma \cup \mathcal{F}^! \cup \mathcal{D}$ appended as a sibling to $x$) is chosen arbitrarily. (Observe that constraints on the results of external calls can be imposed by $\Delta$.)

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 $a_{Bill}$ contains the result of the argument query of `!Bill` (see Figure 3). The dotted arrow indicates the function *eval*.

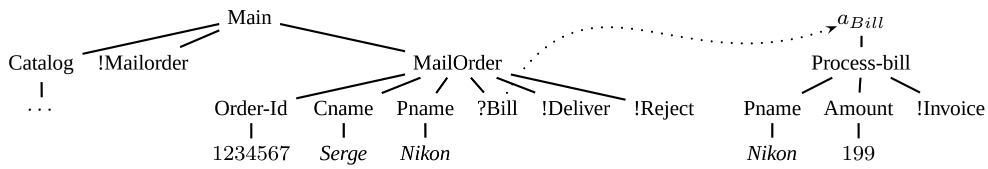



Figure 4: An instance with an $eval$ link

An *initial* instance of $S$ is an instance of $S$ consisting of a single tree whose root is not a function call and that contains 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 *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 and 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 QPattern (for quantified pattern). A *QPattern* is an expression $P(\bar{X})$ where $P$ is a pattern and $\bar{X}$ is a subset of of its variables that are designated as *free*. All other variables are taken to be existentially quantified, locally to $P$ (this could be made explicit by writing $\exists \bar{Y}(P)$, where $\bar{Y}$ is the set of variables occurring in $P$ and not $\bar{X}$.) The syntax of Tree-LTL formulas is defined by the following grammar:

$$\varphi := \texttt{QPattern} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi\, \mathbf{U}\, \varphi \mid \ \mathbf{X}\varphi$$

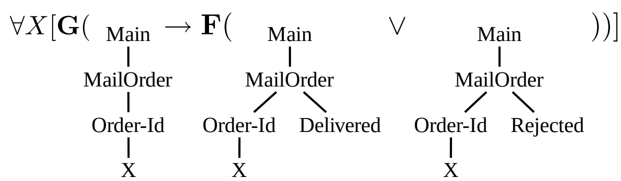
where **U** stands for *until* and **X** for *next*, with the usual semantics, e.g. see [17]. Given a Tree-LTL formula $\varphi$, its free variables are the free variables of its patterns. A Tree-LTL sentence is an expression $\psi = \forall \bar{X} \varphi(\bar{X})$, where $\varphi$ is a Tree-LTL formula and $\bar{X}$ are the free variables of $\varphi$. (As previously mentioned, variables that are not free are existentially quantified locally to each pattern.) We refer to $\bar{X}$ as the *global variables* of $\psi$ (if $\bar{X}$ is empty, we say that $\psi$ has no global variables).

Whenever convenient, we use as shorthand additional standard 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 $\forall \bar{X} \varphi(\bar{X})$ holds for a schema $S$ iff $\varphi(\bar{X})$ holds on every valid run of $S$ with every interpretation of $\bar{X}$ 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 $\sigma = \forall \bar{X} \varphi(\bar{X})$. For a run $\rho$ of $S$, we say that $\rho$ satisfies $\forall \bar{X} \varphi(\bar{X})$, and denote this by $\rho \models \forall \bar{X} \varphi(\bar{X})$, if $\rho$ satisfies $\varphi(h(\bar{X}))$ for each valuation $h$ of $\bar{X}$ into *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 next.

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




*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):*

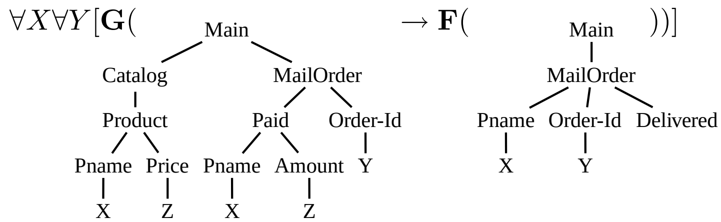



Figure 5: Some Tree-LTL formulas.

The branching-time variants Tree-CTL$^{(*)}$ are defined analogously.

Not surprisingly, satisfaction of Tree-LTL sentences is undecidable for arbitrary GAXML systems. To obtain positive results, we need to place drastic but natural restrictions on these systems. We present in the next section such restrictions and decidability results, and then show how even small relaxations yield undecidability.

# 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 are clearly 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 *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 *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 $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ be a GAXML schema. The *call graph* $G$ of $S$ is a directed graph whose nodes are $\Phi_{\text{int}} \cup \Phi_{\text{ext}}$ and for which there is an edge from $f$ to $g$ if $g \in \mathit{fun}(f)$.

**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.*

Condition (i) is used because recursive DTDs dramatically complicate verification issues. For instance, the satisfisfiability of a Boolean combination of tree pattern queries in presence of recursive DTDs is undecidable [12]. 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. Recall that the initial instance of a run is a single tree. Because of (ii), it therefore includes a bounded number of function calls. Conditions (iii) and (iv) keep the number of service calls made in a run under control by prohibiting the immediate causes of recursion. Condition (ii) deals with another possible source of unbounded calls, the presence of an arbitrary number of them 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, recall that an instance is a forest of trees (except for the initial one); and 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.

Note that, although runs of recursion-free GAXML schemas reach a blocking instance after a bounded number of function calls, they remain infinite-state systems because of the presence of an unbounded number of data values. Thus, there is no straightforward reduction to finite-state model checking.

The main result of the section is that satisfaction of a Tree-LTL sentence by a recursion-free GAXML schema is CO-2NEXPTIME-complete. We first provide the proof of the upper bound, then proceed with the lower bound.

## 3.1 Upper bound

Let $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ be a recursion-free GAXML schema and $\varphi$ a Tree-LTL sentence of the form $\forall \bar{X} \psi(\bar{X})$. Clearly, $S \models \varphi$ iff there is no valid run of $S$ that satisfies $\exists \bar{X} \neg \psi(\bar{X})$. Let $D_{\bar{X}}$ be an arbitrary subset of $\mathcal{D}$ 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 $D \supseteq D_{\bar{X}}$ and no mapping $h$ from $\bar{X}$ to $D_{\bar{X}}$ such that $\rho$ satisfies $\xi = \neg\psi(h(\bar{X}))$, where $\psi(h(\bar{X}))$ 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 $h(Y)$ (note that the resulting $\xi$ has no global variables). Thus, the question of whether $S \models \varphi$ is reduced to a satisfiability problem.

Decidability is shown by proving a small model property. 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 $\xi$ iff its infinite extension satisfies $\xi$. We show that if there is a valid run satisfying $\varphi$ then there is a valid pre-run satisfying $\varphi$ of size bounded by a function computable from $S$ and $\varphi$. The decision procedure is then obtained by guessing a run of that size and checking that it is indeed a valid run satisfying $\varphi$. The following proposition shows that this last step is decidable. Its proof uses standard Büchi automata techniques, after replacing each pattern in $\xi$ by a suitable proposition.

**Proposition 3.2** *Let $S$ be a recursion-free GAXML schema and $\xi$ a Tree-LTL sentence with no global variables. Given a pre-run $\rho = I_0, \dots I_k$ of $S$, one can check whether $\rho$ satisfies $\xi$ using a non-deterministic algorithm in time $O(|\rho|^{|\xi|})$.*

**Proof:** Let $\mathcal{P}$ be the set of tree patterns used in $\xi$. For each $m \in [0, k]$, let $\sigma_m$ be the truth assignment on $\mathcal{P}$ such that for each $P$ in $\mathcal{P}$, $\sigma_m(P) = 1$ iff $I_m \models P$ (note that the latter can be checked in time exponential in $P$). Let $A_\xi$ be the Büchi automaton for the formula $\xi$ where the tree patterns are replaced by distinct propositions (also denoted by $\mathcal{P}$ by slight abuse), and whose alphabet consists of the truth assignments for $\mathcal{P}$. The standard construction of $A_\xi$ produces an automaton whose number of states is exponential in $\xi$. PSPACE algorithm assignment $\sigma$, definition of $A_\xi$) $\rho \models \xi$ iff $A_\xi$ accepts the infinite word $\sigma_0, \dots, \sigma_k, \sigma_k, \dots$, i.e. iff $A_\xi$ 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 $\sigma_0, \dots, \sigma_k$ by reading again $\sigma_k$ at most $2 \cdot |A_\xi|$ times. This yields the desired non-deterministic algorithm taking time $O(|\rho|^{|\xi|})$. □

It remains to show that if there is a valid run satisfying $\varphi$ then there is a valid pre-run of small size. We do this in two steps. In the first step we show that the lenght of a valid pre-run satisfying $\varphi$ can be assumed bounded by an exponential in the size of the schema. In the second step, we show that the size of each instance of the pre-run can also be bounded.

The following proposition takes care of the first step and 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.

**Proposition 3.3** *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.*

**Proof:** Let $\Phi = \Phi_{\text{int}} \cup \Phi_{\text{ext}}$ and $\kappa = |\Phi|$. Let $G$ be the call graph of $S$. By $(iv)$ in the definition of recursion-free schema, $G$ is acyclic. Let $G_0$ be the set of functions $f \in \Phi$ with in-degree zero in $G$. The *depth* of a function $f$ of $\Phi$ is the maximum distance between the node representing $f$ in $G$ and a node of $G_0$. We show by induction on $i$ that a function $f$ of depth $i$ can be called at most $(2 \cdot \kappa)^i$ times. For $i = 0$ this is clear, since 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. Because $S$ is recursion-free, one execution of $g$ may produce at most two direct executions of $f$ (one generated by the argument query, the other by the return query). Hence $f$ is executed at most $2 \cdot |G_f|(2 \cdot \kappa)^{i-1} \leq (2 \cdot \kappa)^i$ as $|G_f| \leq \kappa$. As the depth of $G$ is bounded by $\kappa$, each function is eventually executed at most $(2 \cdot \kappa)^\kappa$ times, hence the bound on the length of the run. □

The next proposition is 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 $\xi$.

**Proposition 3.4** *Let $S$ be a recursion-free GAXML schema and $\xi$ a Tree-LTL sentence with no global variables. 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 doubly exponential in $\xi$ and $S$.*

**Proof:** 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 tags $a_f$) 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 $I_k$ 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 lemmas. 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 $I = (\mathcal{T}, \textit{eval})$ be an instance of $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$. A *sub-instance of* $I$ is an instance $J = (\mathcal{T}', \textit{eval}')$ of $S$ such that (i) each tree $T'$ in $\mathcal{T}'$ is a prefix[2] of some tree $T$ in $\mathcal{T}$, (ii) $\mathcal{T}'$ includes all nodes in $\mathcal{T}$ labeled by symbols $?f$ and $a_f$, and *eval*$'$ maps each node $x$ labeled $?f$ to the tree in $\mathcal{T}'$ that is a prefix of *eval*$(x)$. We denote by $J \sqsubseteq I$ the fact that $J$ is a sub-instance of $I$. Note that *eval*$'$ is uniquely determined by $\mathcal{T}'$ and $I$. An important property of a subinstance is that it preserves the false patterns: If a pattern does not hold in $I$, then it does not 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-recursiveness.

**Lemma 3.5** *Let $S = (\Phi_{int}, \Phi_{ext}, \Delta)$ be a GAXML schema, $I$ and $I'$ instances of $S$ such that $I \vdash I'$, and let $K \sqsubseteq I'$. Then there exists Pre$(K) \sqsubseteq I$ and $K' \sqsubseteq I'$ such that Pre$(K) \vdash K'$, $K \sqsubseteq K'$, and $|\textit{Pre}(K)| \leq d \cdot g + (d \cdot b + 1) \cdot |K|$, where $d$ is the maximum depth of a tree satisfying $\Delta$, $g$ is the maximum size of a guard and $b$ the maximum size of 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 $x$. Let $\gamma(f)$ be the call guard of $f$. For each pattern $P$ occurring in $\gamma(f)$ that holds in $(I, x)$, let $\mu_P$ be a matching of $P$ into $(I, x)$, 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 $d \cdot |\gamma(f)| \leq d \cdot g$.

If $f$ is an external function, then *Pre*$(K)$ consists of $G$ together with $K$, with the label of $x$ changed from $?f$ to $!f$. Suppose $f$ is an internal function. Let $T$ be the tree in $I'$ with root $r$ labeled $a_f$ resulting from the call. Recall that $T$ consists of $r$ with subtrees resulting from the evaluation of the argument query of $f$, *Body* $\rightarrow$ *Head* on $(I, x)$. For each tree $H$ in *Head*, let $c$ be its constructor node and $H_c$ the corresponding subtree. For each matching $\mu$ of *Body* into $(I, x)$ 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 matchings $\mu$ for which $\mu(H_c)$ intersects $K$ for some $H$ in *Head*. Then the nodes of *Pre*$(K)$ are those of $G$ together with those belonging to both $K$ and $I$, and those occurring in $\{\mu(Body) \mid \mu \in \mathcal{M}\}$, together with their ancestors. Note that $|\textit{Pre}(K)| \leq |K| + d \cdot |\gamma(f)| + d \cdot |Body| \cdot |K| \leq d \cdot g + (d \cdot b + 1) \cdot |K|$. To see that *Pre*$(K)$ satisfies the other conditions of the lemma, note first that *Pre*$(K)$ contains $x$ with label $!f$ (the same as in $I$) and $\gamma(f)$ holds in $(\textit{Pre}(K), x)$. Thus, there exists $K'$ such that *Pre*$(K) \vdash K'$ and $K'$ is obtained from *Pre*$(K)$ by a call to $f$ at node $x$. If $f$ is an external function, $K \sqsubseteq K'$ by construction. If $f$ is an internal function, $K'$ is obtained from *Pre*$(K)$ by evaluating *arg*$(f) =$ *Body* $\rightarrow$ *Head* on $(\textit{Pre}(K), x)$. Since by construction all matchings

[2] 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 $T'$.

of *Body* into $(I, x)$ in $\mathcal{M}$ are also matchings in $(\mathit{Pre}(K), x)$, it easily follows that $K \sqsubseteq K'$ (modulo node renaming). Since $\mathit{Pre}(K) \sqsubseteq I$ and the transitions $I \vdash I'$ and $\mathit{Pre}(K) \vdash K'$ are the result of the same function call, it also follows that $K' \sqsubseteq I'$.

Next, suppose $I'$ is obtained from $I$ by the return of the result of a function call $?f$ at node $x$. Suppose $f$ is an external function. Then $\mathit{Pre}(K)$ 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, $\mathit{Pre}(K)$ is obtained similarly to the above. In this case again, $|\mathit{Pre}(K)| \leq |K| + d \cdot |\rho(f)| + d \cdot |Body| \cdot |K|$ where $\rho(f)$ is the return guard of $f$ and *Body* is the body of the return query of $f$, so $|\mathit{Pre}(K)| \leq d \cdot g + (d \cdot b + 1) \cdot |K|$. The proof that $\mathit{Pre}(K) \vdash K'$ where $K \sqsubseteq K' \sqsubseteq I'$ is similar to the above. $\square$

In constructing our "small" run, we will need to enforce validity of the instances with respect to $\Delta$. To this end, we use the notion of "completion" of an instance. Let $J$ be a sub-instance of $I$. A *completion* $\bar{J}$ of $J$ with respect to $I$ and $\Delta$ is defined as follows. Let $max(\Delta)$ be the maximum integer used in the specification of $\Delta$. First, let $J'$ be obtained by adding to $J$ all subtrees of $I$ rooted at nodes in $J$. Next, $\bar{J}$ is obtained from $J'$ as follows. For each node $x$ of $J$, if $x$ has more than $max(\Delta)$ children in $J'$ that are not in $J$ and have the same label $a \in \Sigma \cup \{!f \mid f \in \Phi_{\text{int}} \cup \Phi_{\text{ext}}\}$, retain $max(\Delta)$ of them and remove from $J'$ the rest (together with their subtrees). Similarly, if $x$ in $J$ has more than $max(\Delta)$ children in $J'$ that are not in $J$ and are labeled by (possibly distinct) data values, retain $max(\Delta)$ of them and remove the rest from $J'$. The following is easily seen. Note that, like Lemma 3.5, the following does not assume non-recursiveness.

**Lemma 3.6** *Let $S$ be a GAXML schema. Suppose $I$ is an instance of $S$, $I \models \Delta$, $J$ is a sub-instance of $I$, and $\bar{J}$ is a completion of $J$ with respect to $I$ and $\Delta$. Then for every instance $L$ such that $\bar{J} \sqsubseteq L \sqsubseteq I$, if $x$ is a node in $\bar{J}$, then the set of children of $x$ in $L$ satisfies $\Delta$ (in particular, $\bar{J} \models \Delta$). Furthermore, $|\bar{J}| \leq d \cdot (a \cdot max(\Delta))^d \cdot |J|$, where $d$ is the maximum depth of a tree satisfying $\Delta$ and $a$ is the size of the alphabet of $\Delta$.*

**Proof:** Suppose $\bar{J} \sqsubseteq L \sqsubseteq I$ and $x$ is a node in $\bar{J}$. Consider the children of $x$ in $L$. By construction, for each label $b$, the number of children of $x$ 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 $L$ iff it holds in $I$ for the children of $x$. Since $I \models \Delta$, it follows that $\Delta$ is satisfied by the children of $x$ in $L$. Finally, the bound on $\bar{J}$ is immediate. $\square$

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 $\mathit{Match}_m(\mathcal{P})$ 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 $\bar{J}_m$ of $I_m$ as follows. For the basis, consider $m = k$. Recall that $I_k$ is blocking. For each node $x$ in $I_k$ labeled by a function call $!f$, and each pattern $P$ in $\gamma(f)$ that matches into $(I_k, x)$, let $\sigma_P$ be such a matching. Let $G$ be 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 $\mathit{Match}_k(\mathcal{P})$. Let $\bar{J}_k$ be a completion of $J_k$ with respect to $I_k$ and $\Delta$.

For the inductive step, let $m < k$. Let $\mathit{Pre}(\bar{J}_{m+1})$ be constructed from $\bar{J}_{m+1}$ as in Lemma 3.5. Next, let $J_m$ be the minimum sub-instance of $I_m$ containing $\mathit{Pre}(\bar{J}_{m+1})$, the nodes of $I_m$ that belong to the skeleton of $\rho$, and the nodes in $\mathit{Match}_m(\mathcal{P})$. Finally, let $\bar{J}_m$ be a completion of $J_m$ with respect to $I_m$ and $\Delta$.

We next define by forward induction the desired valid pre-run $R_0, \ldots, R_k$, starting with $R_0 = \bar{J}_0$. As we shall see, $\bar{J}_m \sqsubseteq R_m \sqsubseteq I_m$ for $0 \leq m \leq k$ and $R_m \models \Delta$. The basis ($R_0 = \bar{J}_0$) is clear. Let $0 \leq m < k$ and suppose $R_m$ has been defined, $R_m$ satisfies $\Delta$, and $\bar{J}_m \sqsubseteq R_m \sqsubseteq I_m$. By construction, *Pre*$(\bar{J}_{m+1}) \sqsubseteq \bar{J}_m$, so *Pre*$(\bar{J}_{m+1}) \sqsubseteq R_m$. By Lemma 3.5, *Pre*$(\bar{J}_{m+1}) \vdash K'$ where $\bar{J}_{m+1} \sqsubseteq K' \sqsubseteq I_{m+1}$. Since *Pre*$(\bar{J}_{m+1}) \sqsubseteq R_m \sqsubseteq I_m$, it follows that $R_m \vdash R_{m+1}$ where $K' \sqsubseteq R_{m+1} \sqsubseteq I_{m+1}$, and the transition $R_m \vdash R_{m+1}$ results from the same function call or result return as in $I_m \vdash I_{m+1}$. 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 $F \cap \bar{J}_{m+1}$. In all cases, since $\bar{J}_{m+1} \sqsubseteq K'$, we have the desired inclusions $\bar{J}_{m+1} \sqsubseteq R_{m+1} \sqsubseteq I_{m+1}$.

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 3.6, since $\bar{J}_{m+1} \sqsubseteq R_{m+1}$ and the root belongs to $\bar{J}_{m+1}$.

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 $\bar{J}_{m+1}$). Suppose $f$ is external. Recall that by construction, the answer to the external call consists of sibling subtrees of $\bar{J}_{m+1}$ sitting under some node $x$, and each tree in the answer satisfies $\Delta$ (because $I_{m+1} \models \Delta$). $R_{m+1}$ may contain additional sibling subtrees under $x$ that are not in $\bar{J}_{m+1}$ because they were already in $R_m$, and each satisfies $\Delta$. Since $\bar{J}_{m+1} \sqsubseteq R_{m+1}$, and $x$ is in $\bar{J}_{m+1}$, the set of children of $x$ in $R_{m+1}$ also satisfies $\Delta$, by Lemma 3.6. 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 $\bar{J}_m \sqsubseteq R_m \sqsubseteq I_m$ and each $P \in \mathcal{P}$ that holds in $I_m$ also has a match in $\bar{J}_m$, 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 $R_k$ 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 $l$ 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 3.3).
- The skeleton $sk(\rho)$ of $\rho = I_0, \ldots, I_k$ is bounded by $k \cdot d \cdot 2 \cdot k$. 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 $|sk(\rho)| + d \cdot g \cdot 2 \cdot k + d \cdot |\xi|$. The term $d \cdot g \cdot 2 \cdot k$ bounds the size of $G$ (we need to consider a most $2 \cdot k$ guards) and $d \cdot |\xi|$ bounds the size of *Match*$_k(\mathcal{P})$. Thus, $|J_k| = O(s^2 \cdot l \cdot k^2)$.
- By Lemma 3.6, $|\bar{J}| = O(s^{2s+1} \cdot |J|)$, so $|\bar{J}_k| = O(k^2 \cdot l \cdot s^{2s+3)})$.
- Consider $\bar{J}_m$ for $m < k$. By construction, $|J_m| \leq |Pre(J_{m+1})| + |sk(\rho)| + |$*Match*$_m(\mathcal{P})|$. By Lemma 3.5, $|Pre(J_{m+1})| = O(s^2 \cdot |\bar{J}_{m+1}|)$. Also, $|sk(\rho)| = O(s \cdot k^2)$ ( see above) and $|$*Match*$_m(\mathcal{P})| = O(s \cdot l)$. It follows that $|\bar{J}_m| = O(s^{2s+1}(s^2 \cdot |\bar{J}_{m+1}| + s \cdot k^2 + s \cdot l)) = O(k^2 \cdot l \cdot s^{2s+3} \cdot |\bar{J}_{m+1}|)$.

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

Thus, $\bar{J}_0$ 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 $|\bar{J}_0|^v \cdot h$, 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 $|\bar{J}_m|$, $m \in [0, k]$, to which the bound established above for $|\bar{J}_0|$ applies. Thus, each $R_m$ remains doubly exponential in $S$ and $\xi$. This completes the proof of Proposition 3.4. $\Box$

We are now ready to show the desired upper bound.

**Proposition 3.7** *It is decidable in* CO-2NEXPTIME, *given a recursion-free GAXML schema $S$ and a Tree-LTL sentence $\varphi$, whether each valid run of $S$ satisfies $\varphi$.*

**Proof:** Let $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ be a recursion-free GAXML schema and $\varphi$ a Tree-LTL sentence of the form $\forall \bar{X} \psi(\bar{X})$. In view of Propositions 3.3 - 3.4, a 2NEXPTIME decision procedure for checking whether $S \not\models \varphi$ is the following:

1. Guess $D_{\bar{X}} \subset \mathcal{D}$ with as many elements as variables in $\bar{X}$, and a valuation $h$ of $\bar{X}$ into $D_{\bar{X}}$;
2. Construct the formula $\xi = \neg\psi(h(\bar{X}))$, where $\psi(h(\bar{X}))$ 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 $h(Y)$;
3. Guess a valid initial instance $R_0$ of $S$, of size doubly exponential in $S$ and $\xi$.
4. 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 of size at most doubly exponential in $S$ and $\xi$. A blocking instance $R_k$ is guaranteed to be reached after a number of transitions exponential in $S$.
5. Check that $R_0, \ldots, R_k$ satisfies $\xi$.

Note that (5) remains in 2NEXPTIME by Proposition 3.2. $\Box$

## 3.2 Lower bound

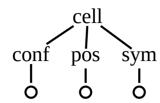
We next establish the lower bound for verification of recursion-free GAXML systems.

**Proposition 3.8** *It is* CO-2NEXPTIME*-hard to check whether a recursion-free GAXML schema satisfies a Tree-LTL sentence.*

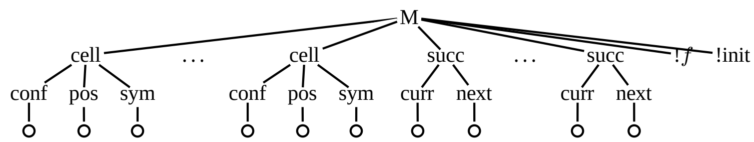
**Proof:** We prove a stronger version of the theorem, in which the Tree-LTL sentence is the fixed sentence *false*, and $S$ is constructed so that the call and return guards of all functions are *true*. (By slight abuse, we say in this case that $S$ has no call guards.) Thus, the necessary control is achieved exclusively using the DTD and data constraints. This will be useful in proving other lower bounds in the paper.

Let $M$ be a non-deterministic Turing Machine running in time $2^{2^n}$ on inputs of size $n$. Let $w$ be a string of length $n$. We construct a a recursion-free GAXML service $S$ such that $M$ accepts $w$ iff $S$ violates *false*. Note that $S$ violates *false* iff $S$ has some valid run.

We next describe the encoding of a computation of $M$ in the initial instance of $S$. A computation of $M$ on input $w$ of length $n$ consists of $2^{2^n}$ successive configurations, each of which is a sequence of symbols of length up to $2^{2^n}$. To identify configurations and positions within each configuration, we use a totally ordered set of $2^{2^n}$ data values. We represent a computation by a set of cells holding a tape symbol (representing also the current state and position of the head) and indexed by a pair consisting of a configuration identifier and a position identifier. We use two constant data values, $\alpha$ and $\beta$ ($\alpha \neq \beta$), to denote the minimum and maximum index. Thus, a cell is represented by a tree of the following form (the circles stand for data values):




The initial instance of $S$ has the following structure:

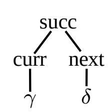



The role of $f$ will become apparent shortly. The role of the function *init* is to enforce the initialization of the computation. The constraints require that ?*init* be present whenever another running call exists. Note that this means that !*init* has to be present in the initial instance of every valid run, and must be the first function to be called. Also, ?*init* has to be present until the end of each valid run. The DTD enforces the above structure whenever !*init* is present, so for the initial instance.

Additionally, we use data constraints to enforce that

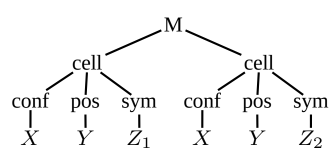
(i) the pair of values of *conf* and *pos* uniquely identify the subtree rooted at *cell*, and

(ii) in the graph *succ* whose edges are the pairs $(\gamma, \delta)$ for which the tree




occurs in the instance, all nodes have in-degree and out-degree at most one. Furthermore, $\alpha$ has in-degree zero and $\beta$ has out-degree zero.

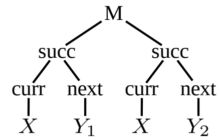
To enforce (i) we use a data constraint forbidding the pattern:




$Z_1 \neq Z_2$

Because trees are reduced, this implies that there are no distinct trees rooted at *cell* and having identical values for *conf* and *pos*, since such trees would have to be isomorphic and thus merged.

Enforcing (ii) is also done by forbidding some patterns. For example, forbidding the following pattern ensures that all nodes have out-degree at most one:




$Y_1 \neq Y_2$

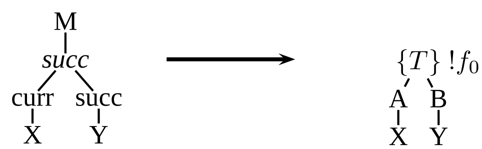
The main steps in the construction are as follows. For brevity, we write *conf* = $\delta$ to mean that the data value under *conf* is $\delta$, and similarly for *pos, sym, curr, next*.

1. Compute the transitive closure of *succ* and check that there is a path from $\alpha$ to $\beta$ of length exactly $2^{2^n}$. Let $P_{\alpha\beta}$ denote the set of nodes along this path (because of $(ii)$, such a path is unique if it exists).

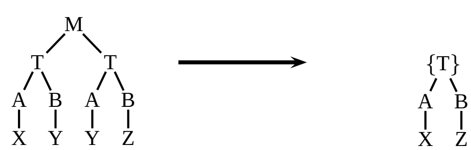
2. Check that, for each $\gamma, \delta \in P_{\alpha\beta}$, there exists a cell for which *conf* $= \gamma$ and *pos* $= \delta$.

3. Verify that, for each $\gamma \in P_{\alpha\beta}$, $\gamma \neq \beta$, if $\delta$ is the successor of $\gamma$, then the configuration of $M$ corresponding to the sequence of cells for which *conf* $= \delta$ is a valid successor to the configuration for which *conf* $= \gamma$. Finally, check that the last configuration (for which *conf* = $\beta$) is accepting.

We now provide more details. For $(1)$, we begin by copying *succ* under a new root $T$ using function $f$. The input query of $f$ is:



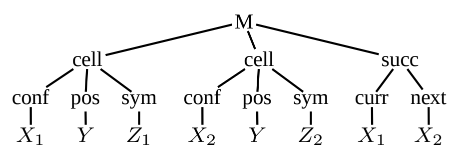

The return query of $f$ simply returns back the result of its input query. The role of $f_0$ is to trigger the computation of the transitive closure of $T$. The computation uses $2n$ additional functions $f_i, f_i'$, $1 \leq i \leq n$, whose purpose is to trigger $2^n$ calls to a function $f_n$ that implements one step in the computation of the transitive closure of the initial $T$. The constraints ensure that $!f$ and $?f$ no longer occur if $?f_0$ occurs, and $f_0$ returns as answer the forest consisting of the two calls $!f_1!f_1'$. Next, for each $i, 1 \leq i < n$, $f_i$ and $f_i'$ return $!f_{i+1}!f_{i+1}'$. Finally, $!f_n'$ returns $!f_n$. Clearly, this results in $2^n$ calls to $f_n$. The call guard of $f_n$ ensures that $?f_n$ is not present in the tree (the calls to $f_n$ have to be done successively), and the input query of $f_n$ is:




The return query returns the result of the input query. Because of the double recursion in $T$, the $2^n$ calls to $f_n$ compute the pairs of nodes at distance at most $2^{2^n}$ in *succ*. The fact that there is a path from $\alpha$ to $\beta$ of length exactly $2^{2^n}$ is checked by data constraints stating that $(\alpha, \beta)$ occur in $T$, but not before the last call to $f_n$ has been made. This is done by requiring that $(\alpha, \beta)$ occur in $T$ when none of the functions used so far is present (except for ?*init*), and that $(\alpha, \beta)$ not occur in $T$ if such functions are present.

We next proceed with (2). The idea is as follows. For each configuration identifier $\gamma$, we extract the sub-graph *succ*$_\gamma$ of *succ* using only nodes $\delta$ for which there is a cell with *conf* $= \gamma$ and *pos* $= \delta$. We then compute, for each $\gamma$, the transitive closure of *succ*$_\gamma$, similarly to the above, using $2^n$ new functions (with the difference that $\gamma$ is carried as a parameter in the computation of the transitive closure). Next, we collect all $\gamma$ for which $(\alpha, \beta)$ belongs to the transitive closure of *succ*$_\gamma$, in a new forest of trees with root *OK*. Finally, we select the sub-graph of *succ* using only the values collected under *OK*, compute its transitive closure as above, and check that $(\alpha, \beta)$ belongs to it. This guarantees that (2) holds. Thus, the *cell* trees for which the values of *conf* and *pos* are both in $P_{\alpha\beta}$ provide a valid representation of $2^{2^n}$ configurations, each of length $2^{2^n}$.

It remains to verify (3). It is easy to enforce, using data constraints of size polynomial in $M$ and $w$, that all values of *sym* are tape symbols and that the initial configuration (given by the cells for which *conf* $= \alpha$ and for which the value of *pos* is in $P_{\alpha\beta}$) contains $w$. It is also easy to check that the final configuration is accepting. It remains to verify that consecutive configurations result from valid transitions of $M$. Recall that a single move of a Turing machine may only affect the cell pointed to by the head, and its left or right neighbor. Call these the *neighbors* of the head. There are two cases to consider. First, the contents of cells that are not neighbors of the head must remain unchanged. Second, neighbors of the head must change according to a valid move of $M$. Both cases can be easily taken care of by data constraints forbidding patterns that violate the requirements. For example, to say that the content of the cell at position $Y$ remains unchanged in the transition from configuration $X_1$ to configuration $X_2$, we forbid the following pattern:




$Z_1 \neq Z_2$

We omit the straightforward details.

The sequencing of the steps described above is easily enforced using data constraints. Since distinct functions are used in different steps, completion of a stage can be detected by the absence of function symbols involved in that and previous stages. Thus, to enforce that a function $h$ is triggered at a given stage, it is sufficient for the data constraints to require the absence of function symbols from previous stages when $?h$ is present. Finally, a data constraint can check that the last configuration of $M$ is accepting. This completes the construction of $S$, which is clearly polynomial in $M$ and $w$.

It is now easy to see that $M$ accepts $w$ iff there exists some valid run of $S$. The "only-if" part is obvious. For the "if" part, recall that the guards of all functions are *true*, so every run reaches a blocking instance with no remaining functions. If in addition the run is valid, then the DTD and constraints specified above are satisfied throughout the run, and the run constitutes a full, correct simulation of an accepting computation of $M$ on input $w$.

Equivalently, $M$ accepts $w$ iff $S$ violates *false*. $\Box$

In the above proof, we placed most of the burden of control on the data constraints, and used a trivial Tree-LTL sentence. Conversely, it is possible to shift the onus of the simulation from the data constraints to the Tree-LTL formula. This points to an interesting trade-off between data constraints and temporal formulas, summarized below.

**Corollary 3.9** $(i)$ *It is* 2NEXPTIME-*hard to check, given a recursion-freeGAXML schema $S$ with no guards, whether $S$ has some valid run (or equivalently, $S \not\models$* false*).* $(ii)$ *It is* CO-2NEXPTIME-*hard to check, given a* *recursion-free GAXML schema $S$ with no data constraints and no guards, and a Tree-LTL sentence $\varphi$ with no global variables, whether $S \models \varphi$.*

**Proof:** First, $(i)$ follows immediately from the proof of Proposition 3.8. Consider $(ii)$. This follows from the fact that the data constraints of a schema can be absorbed into the Tree-LTL sentence to be verified. Indeed, let $\sigma$ be the data constraint of $S$ constructed in the proof of Proposition 3.8. Let $S'$ be identical to $S$, but without data constraints. Clearly, $S \models$ *false* iff $S' \models \neg(\mathbf{G}\ \sigma)$. Since $\sigma$ is a Boolean combination of tree patterns with no free variables, this is a syntactically correct Tree-LTL sentence with no global variables. □

Propositions 3.7 and 3.8 yield the main result of the section.

**Theorem 3.10** *It is* CO-2NEXPTIME*-complete to decide, given a recursion-free GAXML schema $S$ and a Tree-LTL sentence $\varphi$, whether each valid run of $S$ satisfies $\varphi$.*

**Remark 3.11** *While the worst-case* CO-2NEXPTIME *complexity of verification we have just shown may appear daunting, the complexity is likely to be much lower in many practical situations. For example, one cause of the high complexity is the fact that, in a recursion-free GAXML schema, the length of runs can be exponential in the number of functions of the schema. However, this requires a rather convoluted use of the functions. In many practical situations, the length of runs is only linear in the number of functions. For instance, this happens under restrictions such as the following:*

*(i) the call graph of the schema is a tree, and*

*(ii) if a function $f$ passes a call $!g$ in its argument query, its return query does not contain the same call $!g$.*

*Such conditions are satisfied naturally when functions model a hierarchical set of tasks, and can be easily checked syntactically. It is straightforwardly to see, by revisiting the proofs of Propositions 3.7 and 3.8, that under conditions (i) and (ii) the complexity of verification for recursion-free GAXML schemas and Tree-LTL properties becomes* CO-NEXPTIME*-complete. Within the broader landscape of static analysis, this complexity is quite reasonable. For instance, recall that even satisfiability of Barnays-Schönfinkel FO sentences, a much simpler question, already has complexity* NEXPTIME *[11]. To bring the complexity down to* PSPACE*, one would have to impose more drastic restrictions. For example, the complexity of verification is* PSPACE *if, in addition to (i) and (ii), we bound by a constant the number of functions of the schema and the maximum depth of trees allowed by the DTD.*

Using techniques similar to the above, we can show decidability of some other useful static analysis tasks for recursion-free GAXML. We first consider successful termination, then typechecking. We say that a run terminates successfully iff its blocking instance has no pending running calls.

**Theorem 3.12** (**Successful termination**) *It is* CO-2NEXPTIME*-complete whether, for a given recursion-free GAXML schema $S = (\Phi_{int}, \Phi_{ext}, \Delta)$, each valid run of $S$ ends in a blocking instance with no running function calls.*

**Proof:** For the upper bound, successful termination can be reduced to satisfaction of a Tree-LTL sentence by a recursion-free GAXML schema. For successful termination, the property to be verified is

$$\mathbf{F}[\nu\ \wedge \bigwedge_{f \in \Phi_{\text{int}} \cup \Phi_{\text{ext}}} \neg\, \gamma'(f)]$$

where $\nu$ 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 $!f$. 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 the lower bound, consider Corollary 3.9. Based on the proof of Proposition 3.8, using a simulation of a 2NEXPTIME Turing machine $M$ on input $w$, it was shown there that it is 2NEXPTIME-hard whether a recursion-free GAXML schema $S$ has a valid run. Recall that the construction uses a function *init* that must be fired first in every valid run. We modify slightly the specification by having the return guard of *init* be *false*. Thus, the running call ?*init* is present in the blocking instance of every valid run, so successful termination is violated. It follows that $M$ accepts $w$ iff $S$ violates the successful termination property. This proves the CO-2NEXPTIME lower bound. □

We next consider typechecking. Let $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ be a GAXML schema. We say that $S$ *typechecks* with respect to $\Delta$ if for every run of $S$, if the initial instance satisfies $\Delta$, then every instance in the run satisfies $\Delta$.

**Theorem 3.13** (**Typechecking**) *It is* CO-2NEXPTIME*-complete whether a recursion-free GAXML schema* $S = (\Phi_{\mathit{int}}, \Phi_{\mathit{ext}}, \Delta)$*, typechecks with respect to* $\Delta$*.*

**Proof:** The proof of the upper bound is analogous to that of Proposition 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 $\Delta$, $I_k$ satisfies $\Delta'$. Suppose, to the contrary, that there exists $\rho = I_0, \ldots, I_k$ such that $I_0$ is an initial instance (satisfying $\Delta$), $I_i \vdash I_{i+1}$ for $0 \leq i < k$, 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 doubly exponential in $S$. The construction is similar to that in the proof of Proposition 3.4. This shows that checking the existence of a violation of typechecking with respect to the DTD $\Delta'$ can be done in 2NEXPTIME, so typechecking with respect to $\Delta'$ is in CO-2NEXPTIME. 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 $S' = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta')$, and check that every valid pre-run of $S'$ satisfies the Tree-LTL property $\psi \rightarrow \mathbf{G}\, \psi$. This can be done in CO-2NEXPTIME by Theorem 3.10. Thus, typechecking is decidable in CO-2NEXPTIME.

Now consider the lower bound. We use again the proof of Proposition 3.8 and Corollary 3.9 (i). Recall that the proof constructs from a 2NEXPTIME Turing machine $M$ and input $w$ a recursion-free GAXML schema $S$ with no guards that has a valid run iff $M$ accepts $w$. The static constraints of $S$ consist of a DTD $\Delta'$ and a conjunction $\psi$ of data constraints. It can be seen that all data constraints in $\psi$ can be simulated by appropriate call guards for the functions of $S$. Thus, we construct a modified schema $S'$ by replacing the data constraints with call guards. In addition, let $\xi$ be a new data constraint checking the existence of some function call in the instance. Let the static constraints of $S'$ consist of $\Delta' \cup \{\xi\}$. Note that $\xi$ is violated by a run of $S'$ iff its blocking instance has no function calls. Clearly, this happens iff the run successfully checks that the initial instance encodes an accepting computation of $M$ on $w$. Thus, $S'$ typechecks with respect to $\Delta' \cup \{\xi\}$ iff $M$ does not accept $w$. This proves the CO-2NEXPTIME lower bound. □

**Remark 3.14** *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 with respect to* $\Delta$*). A more lenient variant would typecheck subject to the* assumption *that results from calls to external functions do not cause violations. Theorem 3.13 can be easily extended to this variant. In particular, note that the lower bound holds even with no external functions.*

# 4 Beyond recursion-free schemas

In this section we prove that decidability of satisfaction of a Tree-LTL formula by a GAXML 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.

Several proofs 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 $[\bar{A}] \subseteq [\bar{B}]$ where $\bar{A}$ and $\bar{B}$ are sets of attributes of $R$ of the same size. $R$ satisfies $[\bar{A}] \subseteq [\bar{B}]$ if $\pi_{\bar{A}}(R) \subseteq \pi_{\bar{B}}(R)$. A functional dependency (FD) over $R$ is an expression $V \to C$, where $V$ is a set of attributes and $C$ an attribute of $R$. Relation $R$ satisfies $V \to C$ 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 $F$.

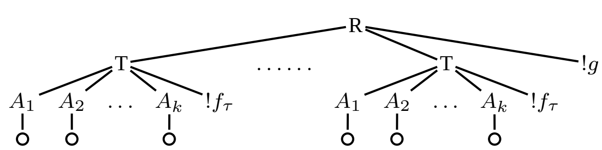
For (1), undecidability is a simple consequence of the fact that satisfiability of Boolean combination of patterns in the presence of a DTD is already undecidable [12]. The first result concerns extensions (2-3). We prove a strong undecidability result, showing that even reachability of an instance satisfying 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.

**Theorem 4.1** *It is undecidable, given a positive pattern $P$ without variables and a GAXML 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 $S$.*

**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 $R$. We construct a GAXML schema $S$ satisfying the stated restrictions, and a tree pattern $P$, such that $\Gamma \not\models F$ iff some instance satisfying $P$ is reachable in a run of $S$. We represent relation $R$ with attributes $A_1 \dots A_k$ in the standard way, as a tree described by the DTD[3]:

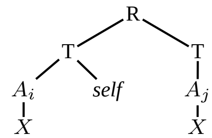
$$\begin{array}{rcl} R & \to & T^* \\ T & \to & A_1 \dots A_k \\ A_i & \to & dom \end{array}$$

Consider (2). 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 $f_\tau$ 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 *false*, so no answer is ever returned. We add one node labeled $!f_\tau$ 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:



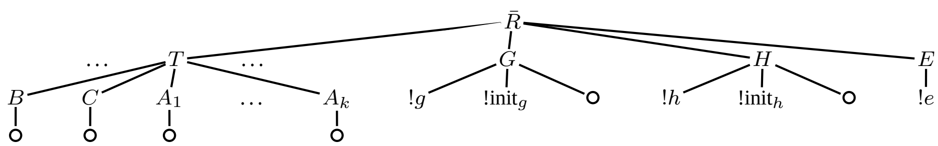

[3]This classical notation maps in the obvious way to constraints in our DTDs.

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




The guard $\gamma(g)$ of $g$ is the conjunction of several tree patterns. The first simply checks that no node labeled $!f_\tau$ exists in the tree. This ensures that all calls to the $!f_\tau$ have been made, which implies that their guards were true, so no $\tau \in \Gamma$ is violated. Satisfaction of the FDs 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 $\Gamma \not\models F$.

Next, consider (3). Suppose $S$ 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 FDs 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 $R$. We denote the relation $R$ augmented with attributes $B, C$ by $\bar{R}$. The IDs are checked by stepping through the tuples of $\bar{R}$ 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 $\bar{R}$ is represented below, together with some additional structure.




Specifically, we add continuous functions $g$, $h$ and $e$, with respective parents $G$, $H$ and $E$, all under root $\bar{R}$. For technical reasons we need two additional functions $\texttt{init}_g$ and $\texttt{init}_h$ that appears under $G$ and $H$. The DTD rules for $G$,$H$ and $E$ are:

$$\begin{array}{lcl} G & \rightarrow & (!g \ + \ ?g)(!\texttt{init}_g \ \textit{dom} \ + \ ?\texttt{init}_g \ \textit{dom}^*) \\ H & \rightarrow & (!h \ + \ ?h)(!\texttt{init}_h \ \textit{dom} \ + \ ?\texttt{init}_h \ \textit{dom}^*) \\ E & \rightarrow & !e \ + \ ?e \end{array}$$

The role of $\texttt{init}_g$ (similarly for $\texttt{init}_h$) is simply to enforce the presence of a unique data value under $G$ and $H$ in the initial instance. Multiple data values may appear once $\texttt{init}_g$ has been called. We define the call guard of $\texttt{init}_g$ to be *true* and its return guard *false*, so that no answer is ever returned. An initial instance of $S$ has the shape above.

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*, distinct data values $\alpha$ and $\beta$. Similarly to the proof of Proposition 3.8, we can use data constraints to enforce that:

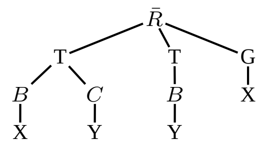
(i) $B$ and $C$ are keys for relation $\bar{R}$ represented by the sub-trees rooted at $T$;

(ii) $\alpha$ and $\beta$ each occur under some $B$, $\alpha$ has in-degree zero and $\beta$ has out-degree zero in $G_{BC}$;

(iii) $\alpha$ occurs under $G$ and under $H$ (so it is the unique data value under these nodes in the initial instance).

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 $\alpha$ in $G_{BC}$. The guard of $g$ is *true*, and its argument query has head $\{Y\}$ and body

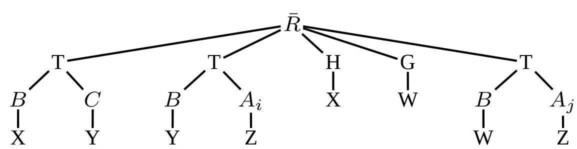



Its return guard is *true* and the return query simply returns the result of the argument query.

The guard of $h$ checks the following:

- $\beta$ occurs under $G$ (so $g$ has been called at least once and $\beta$ is reachable from $\alpha$ in $G_{BC}$). Because of (i) and (ii) above, if this holds, then there is a unique simple path from $\alpha$ to $\beta$ in $G_{BC}$; the data values under $G$ are exactly the nodes along this path;

Let $R_{\alpha\beta}$ denote the sub-relation of $R$ consisting of the tuples indexed by data values along the path from $\alpha$ to $\beta$ (available under $G$). The role of $h$ is to check that $R_{\alpha\beta}$ satisfies all the IDs of $\Gamma$. To this end, $h$ re-does the computation performed by $!g$, with the additional task of checking, as soon as a data value $Y$ reachable from $\alpha$ is detected, that the tuple of $R_{\alpha\beta}$ indexed by $Y$ satisfies all IDs of $\Gamma$ within $R_{\alpha\beta}$. 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 $R[A_i] \subseteq R[A_j]$, the body of the argument query of $!h$ is augmented as follows:




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

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

Thus, the guard of $e$ becomes true iff $R_{\alpha\beta}$ 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 $S$ iff $\Gamma \not\models F$. $\square$

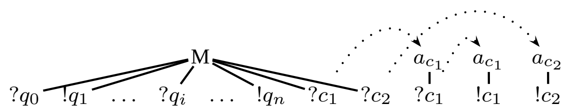
One can use a similar proof to show that Condition (4) also yields undecidability. Instead, we use the fact that, with cyclic call graphs, we can generate arbitrarily long sequences of running function calls allowing us to code two-counter automata. The interest of this alternative proof is that the result holds even in absence of data values. Indeed, as we will see, cyclic call graphs are much more powerful that continuous functions.

**Theorem 4.2** *It is undecidable, given a positive pattern $P$ without variables and a GAXML 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 $S$.*

**Proof:** 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 [21].

Let $M$ be a deterministic counter machine whose set of states is $Q = \{q_0, \ldots, q_n\}$ and counters $C_1, C_2$. Let $q_0$ be the initial state. We assume wlog that $q_0$ is never used again in the course of the computation. We may also assume that both counters are incremented or decremented at each move.

We next describe a GAXML system $S$ simulating $M$. We begin with a sketch of the main idea, then provide more details. For each $q \in Q - \{q_0\}$ we use one function denoted (by slight abuse) $q$. Intuitively, the presence of a running call $?q$ indicates that $M$ is in state $q$. We also use functions $c_1, c_2$ to represent the two counters. The value of counter $C_i$ is the number of active running calls to $c_i$ in the instance. For instance the configuration where $M$ is in state $q_i$ and $C_1 = 2$ and $C_2 = 1$ will correspond to the presence of the following pattern in the document:




We use the DTD to enforce that, throughout the simulation, there is at most one running call to a function $q$ (excepting $?q_0$).

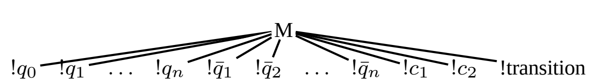
In order to code transitions between configurations we use extra auxiliary functions. For each $q \in Q - \{q_0\}$ we use one function denoted $\bar{q}$. Intuitively a running call $?\bar{q}$ indicates that $M$ should switch to state $q$. We also have functions *inc*$_1$, *inc*$_2$, *dec*$_1$, *dec*$_2$ whose role is to indicate that the corresponding counter should be incremented or decremented. Typically a call to $c_i$ will produce an occurrence of *inc*$_i$ while a return from $c_i$ will produce an occurrence of *dec*$_i$. Again we use the DTD to enforce that, throughout the simulation, there is at most one running call to a function $\bar{q}$.

In order to check that the transition is correct we use a function *transition* whose guard checks that the choice for the next state and updates to the counters are done according to the transition rules of $M$. For instance, if $M$ in state $p$ moves to state $q$ while incrementing the counter $C_1$ and decrementing the counter $C_2$, the guard checks that $?p$, $?\bar{q}$ and *inc*$_1$ and *dec*$_2$ are present in the tree.

Finally, there is a cleaning phase that removes all intermediate function calls and triggers $!q$.

Implementing the above sequence requires some careful control achieved by a few additional functions together with the DTD and data constraints (using only structural information). We now provide the details. To concisely describe the DTD, we use as a notational convention Boolean combinations of symbols, where $a$ stands for $|a| > 0$. The instances of $S$ have depth one (trees consist of a root and their children, with no data values). The root is assumed to be $M$, unless otherwise specified.

We first enforce the initialization of the simulation to the start configuration of $M$. To this end, the DTD includes the constraint stating $!q_0 \vee ?q_0$, and that, if $!q_0$ is present, then the instance has the form



The return guard of $q_0$ is *false*, so the call $?q_0$ never disappears (but is henceforth ignored). The call guards of all functions $q$ other than $q_0$ require the presence of $?q_0$. Thus, the only call that can be made in the initial instance is to $q_0$. This produces the representation of the start configuration of $M$ (with the additional function *transition*).

As described above, the simulation next loops through two main stages:

1. perform a transition: designate the next state $q$ by calling $\bar{q}$, and increment or decrement the counters;

2. reset the states: update the current state to $q$ and reset $?\bar{q}$ to $!\bar{q}$.

To control the computation, we use two functions to identify the above stages: *transition* for (1), and *reset* for (2). A call to *transition* returns as answer !*reset* and a call to *reset* returns as answer !*transition*.

The transition stage (1) proceeds as follows. Initially, the instance contains at the beginning of this stage at most one function call $?p$ different from $?q_0$ and no calls of the form $?\bar{q}$, and there are zero or more running calls to $c_1$ and $c_2$. We first arbitrarily trigger $!\bar{q}$ and increment or decrement the counters, then enforce correctness of the move using the call guard of !*transition*. The call guard of the functions $\bar{q}$ checks that !*transition* is present. The argument queries of $\bar{q}$ are empty, their return guards *true* and the answer consists of $!\bar{q}$. Note that multiple calls to such functions can be made during one transition phase. This is harmless, since correctness is only checked for the last call made before the firing of !*transition*, which then remains fixed until the reset stage. On the other hand, the current state must be fixed during the transition stage. This is ensured by having the call guard of each $q$ require the presence of ?*transition*. The argument queries of $q$ are empty, their return guards *true* for $q \neq q_0$, and the answer consists of $!q$.

The counters are incremented or decremented non-deterministically as follows.

To increment $C_i$, the function $!c_i$ is called (its guard is specified below). Its argument query produces the forest

$$!c_i \;\; !inc_i \;\; !ok_i$$

The function call !*inc*$_i$ signals that counter $C_i$ has been incremented. The role of the function *ok*$_i$ is to check that !*transition* is present using its call guard (recall that this cannot be done by return guards, whose scope is the local tree of the running call). Thus, the call guard of *ok*$_i$ checks for the presence of !*transition* and of $!c_i$ as a sibling (the latter ensures that !*ok*$_i$ may only fire in the last running call of $c_i$). Its return guard is *true*, and the answer is empty. The guard of $!c_i$ checks for the presence of !*ok*$_i$ as a sibling, and of !*transition* in the global instance. It also checks the absence of $!inc_i$ and $!dec_i$ (this prevents repeated increment or decrement of the counter in a single transition).

To decrement $C_i$, the answer to the most recent call $?c_i$ is returned, consisting of the forest

$$!c_i \;\; !dec_i$$

where $!dec_i$ is meant to signal that counter $C_i$ has been decremented. To prevent multiple updates of $C_i$ in a single transition, the return guard of $c_i$ checks the absence of $!inc_i$ and $!dec_i$. Additionally, to make sure the decrement occurs only during the transition phase, the return guard also checks the absence of !*ok*$_i$ or ?*ok*$_i$ (implying that !*ok*$_i$ was fired so the call guard of !*ok*$_i$ is true). The functions $inc_i$ and $dec_i$ both have call guard ?*transition*, return guard *true*, and return the empty forest as answer. This ensures that the "locks" $!inc_i$ and $!dec_i$ are not released before the transition stage is over, which is also needed to prevent multiple increments or decrements in the same transition. The correctness of the transition (next state and counter updates) is ensured by the call guard of !*transition*, specifying the correct combinations of current state, next state, whether each counter was zero prior to the update (detected by the presence of $!inc_i$ and of $?c_i$ under

$M$ but not under some $a_{c_i}$), and whether each counter was incremented or decremented (detected by the presence of $!inc_i$ and $!dec_i$).

The move to stage (2) is triggered by a call to !*transition*, returning !*reset*. To ensure that all $inc_i$ and $dec_i$ have been erased, the DTD requires that none of $!inc_i, ?inc_i, !dec_i, ?dec_i$ occurs when !*reset* is present. This makes it possible for the counters to be updated again during the next transition. The state reset is carried out as follows. The fact that the current state is set to $q$ where $\bar{q}$ is the next state is ensured by requiring in the DTD that $?q$ and $?\bar{q}$ both occur for some $q \in Q - \{q_0\}$ when !*reset* is present. The answer to *reset* is !*transition*, and the DTD requires that no $?\bar{q}$ occurs when !*transition* is present (so the answer to $?\bar{q}$ must be returned while ?*reset* is present). This completes the loop. The following table summarizes the progression of one such loop (starting from state $p$) correlated to the evolution of the "control flags" !*transition* $\rightarrow$ ?*transition* $\rightarrow$ !*reset* $\rightarrow$ ?*reset* $\rightarrow$ !*transition*:

| flag | event |
|---|---|
| !*transition* | fire $!\bar{q}$ and update counters |
| ?*transition* | remove $inc_i$ and $dec_i$<br>reset $?p$ to $!p$, fire $!q$ |
| !*reset* | $?q$ and $?\bar{q}$ are both present |
| ?*reset* | reset $?\bar{q}$ to $!\bar{q}$ |
| !*transition* | $q$ is new current state |

It can be seen that the $S$ thus constructed simulates $M$. Finally, let $P$ be the positive pattern checking the presence of $?r$ for $r \in Q$. It is clear that $M$ reaches state $r$ iff an instance satisfying $P$ can be reached in a 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 GAXML 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 GAXML 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 acyclic 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 subpatterns. 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: $r[/a/X][\neg\ /b/X]$. Undecidability is again shown using a reduction from the implication problem for FDs and IDs.

**Theorem 4.4** *It is undecidable, given a positive pattern $P$ without variables, and a recursion-free GAXML 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 $S$.*

**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 $?g$. $\Box$

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 GAXML 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 $\exists \bar{X} \xi(\bar{X})$.

**Theorem 4.5** *It is undecidable, given a recursion-free GAXML schema $S$ and a $\exists$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 a $\exists$Tree-LTL sentence $\varphi = \exists \bar{X}(\neg\xi(\bar{X}))$ such that $S \models \varphi$ iff $\Gamma \models F$. Equivalently, we show that there exists a run of $S$ satisfying $\forall \bar{X} \xi(\bar{X})$ iff $\Gamma \not\models F$. We represent relation $R$ as in the proof of Theorem 4.1, 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 $\tau = [\bar{A}] \subseteq [\bar{B}]$ of $\Gamma$, $\xi$ contains a conjunct stating that, if $\bar{X}_\tau$ is a tuple in $R$, then $\pi_{\bar{A}}(\bar{X}_\tau) \subseteq \pi_{\bar{B}}(\bar{X}_\tau)$. This can be expressed by tree patterns with free variables $\bar{X}_\tau$. Finally, $\xi$ includes the conjunct **X** $(//?g)$ stating that $g$ is called in the first transition. Let $\bar{X}$ consist of all the variables occurring in $X_\tau$, for $\tau \in \Gamma$. Clearly, $\forall \bar{X} \xi(\bar{X})$ is satisfied in a run of $S$ iff the relation $R$ represented in the initial instance satisfies $\Gamma$ and violates $F$, i.e. $\Gamma \not\models F$. $\Box$

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 **E** 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 GAXML schema $S$, if $S$ satisfies*[4] **AXEG** $(\neg P)$.

**Proof:** We prove, equivalently, that it is undecidable whether there exists an initial instance of $S$ satisfying **AF**$P$. 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 $\Gamma \not\models F$ iff 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

[4] We assume a unique start state from which there is a transition to each initial instance of $S$.

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 $F$. $\square$

**Decidability** As promised, we now exhibit several useful verification tasks that remain decidable even for recursive GAXML 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 with respect to a Boolean combination of patterns. The proof uses a variation of the small model technique developed for showing Proposition 3.7.

**Theorem 4.7** (**Safety**) *It is decidable in* CO-NEXPTIME, *given a GAXML 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 $I \vdash J$, if $I \models \varphi$ then $J \models \varphi$.*

**Proof:** The proof uses Lemmas 3.5 and 3.6. (Recall that they do not assume non-recursiveness.) 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 ($\dagger$) there exists a tree with no nodes labeled $?f$, satisfying $\Delta_0$ and $\psi \wedge \neg\varphi$. 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 ($\dagger$) holds, then there exists a tree $I_0$ with the same property and of size exponential in $|S| + |\varphi|$. Indeed, suppose $I$ satisfies ($\dagger$). 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 $\bar{I}_0$ be the completion of $I_0$ with respect to $\Delta_0$. By Lemma 3.6, $|\bar{I}_0| \leq d \cdot (a \cdot max(\Delta))^d \cdot |I_0|$, 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, $|\bar{I}_0| \leq d^2 \cdot (a \cdot max(\Delta))^d \cdot (|\psi| + |\varphi|)$, and $\bar{I}_0$ is exponential in $|\Delta| + |\varphi|$. Clearly, $\bar{I}_0$ satisfies $\Delta_0$ and $\psi \wedge \neg\varphi$.

Now consider (ii). Once again, we use a small model property. Suppose there exist valid instances $I$ and $J$ of $S$ such that $I \vdash J$, if $I \models \varphi$ but $J \not\models \varphi$. We can show that there exist valid instances $I_0$ and $J_0$ of $S$, of size exponential in $|S| + |\varphi|$, such that $I_0 \models \varphi$ but $J_0 \not\models \varphi$. 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. $\square$

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 3.10 (proof omitted).

**Theorem 4.8** (**Bounded reachability**) *It is decidable in* 2NEXPTIME, *given a GAXML schema $S$, a Boolean combination $\varphi$ of patterns, 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 Compositions of GAXML Systems

We next discuss how our results can be extended to compositions of GAXML systems. Typically, GAXML systems participating in such a composition may be hosted on different peers, so this is generally a multi-peer system. We informally describe a model for GAXML compositions and show that our decidability results extend to such systems.

**Definition 5.1** *A GAXML composition $\mathcal{S}$ is a non-empty set $\{S_i\}_{1\leq i\leq n}$ of GAXML schemas with disjoint sets of internal functions.*

Consider a composition $\mathcal{S} = \{S_i\}_{1\leq i\leq n}$. Let $S_i = (\Phi^i_{\text{int}}, \Phi^i_{\text{ext}}, \Delta_i)$, $1 \leq i \leq n$. Intuitively, each $S_i$ supports the set of functions specified by $\Phi^i_{\text{int}}$. Some of the external functions $\Phi^i_{\text{ext}}$ may be supported by another schema $S_j$ in the composition, in which case they belong to $\Phi^j_{\text{int}}$. Given a composition $\mathcal{S}$ as above, we say that the functions in $\bigcup_{1\leq i\leq n} \Phi^i_{\text{int}}$ are *internal* to $\mathcal{S}$, and the functions in $(\bigcup_{1\leq i\leq n} \Phi^i_{\text{ext}}) - (\bigcup_{1\leq i\leq n} \Phi^i_{\text{int}})$ are *external* to $\mathcal{S}$.

An *instance* $I$ of a GAXML composition $\mathcal{S} = \{S_i\}_{1\leq i\leq n}$ is a pair $(\{\mathcal{T}_i\}_{1\leq i\leq n}, \textit{eval})$ where:

- Each $\mathcal{T}_i$ is a GAXML forest;
- $\mathcal{T}_i$ and $\mathcal{T}_j$ have disjoint sets of nodes for $i \neq j$;
- $eval$ is a bijection associating to each node of $\mathcal{T} = \bigcup_{1\leq i\leq n} \mathcal{T}_i$ labeled by a running function call $?f$, where $f$ is internal to $\mathcal{S}$, a tree in $\mathcal{T}$ whose root is labeled $a_f$.

An instance $(\{\mathcal{T}_i\}_{1\leq i\leq n}, \textit{eval})$ is valid if each $\mathcal{T}_i$ satisfies $\Delta_i$.

An *initial* instance of $\mathcal{S}$ is an instance $(\{\mathcal{T}_i\}_{1\leq i\leq n}, \emptyset)$, where each $(\mathcal{T}_i, \emptyset)$ is an initial instance of $S_i$, $1 \leq i \leq n$. (Thus, $\mathcal{T}_i$ consists of a single tree whose root is not a function call and containing no running calls). The definitions of *run* and valid run of $\mathcal{S}$ are as before, with two differences. First, if $f \in \Phi^i_{\text{int}}$, then all trees whose roots are labeled $a_f$ belong to $\mathcal{T}_i$. The second difference concerns the scope of call guards and queries. Recall that in the single-peer case, all guards and input queries were evaluated within the entire instance $\mathcal{T}$. We now impose some locality. More precisely, if $f \in \Phi^i_{\text{int}} \cup \Phi^i_{\text{ext}}$, the call guard and argument query of $f$ are both evaluated within $\mathcal{T}_i$. The semantics of return guards and queries stays unchanged. Thus, for an internal function $f$, these are evaluated, as before, within the tree rooted at $a_f$ that corresponds to the running call.

Observe that our definition amounts to viewing a composition of GAXML peers as a single GAXML with a separate workspace assigned to each peer. If we think of the content of a peer as its state, the state of the composition is the product of the states of the peers, a classical viewpoint. Consequently, the decidability result of Section 3 carries immediately to compositions, as we will see shortly.

**Remark 5.2** *It should be noted that the above composition model makes strong synchronicity assumptions ensuring that each function call causes simultaneous state transitions in the calling and receiving peers. Such tight synchronization is hard to enforce in real systems. However, this 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 composition model, with explicit messages and queues, requires an extension of our GAXML model. This raises new interesting questions that are left for future work.*

The syntax and semantics of the language Tree-LTL are the same as before. The definition of recursion-free GAXML compositions is the same as for single schemas, where the internal and external functions are taken to be those of $\mathcal{S}$ (rather than those of the individual schemas). As promised, the main decidability result for single schemas extends to compositions.

**Theorem 5.3** *It is* CO-2NEXPTIME*-complete whether a recursion-free GAXML composition satisfies a Tree-LTL sentence.*

**Proof:** The lower bound transfers trivially from the single schema case. For the upper bound, we use an easy reduction to the single schema case. Let $\mathcal{S}$ be a recursion-free GAXML composition and $\varphi$ a Tree-LTL sentence. We can construct in polynomial time a recursion-free GAXML schema $S'$ and a Tree-LTL sentence $\varphi'$ such that $\mathcal{S} \models \varphi$ iff $S' \models \varphi'$. The schema $S'$ is essentially the union of the schemas of $\mathcal{S}$, slightly modified to enforce the locality of call guards and argument queries. This in turn requires minor modifications to $\varphi$, yielding $\varphi'$. We omit the details. $\Box$

To conclude this section, we mention that another model of composition of GAXML systems is considered in [6]. The focus of their work is on the specification of the interface between the systems in the composition.

# 6 Conclusions

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 GAXML systems. As a side effect, they also provide insight into the subtle interplay between the various features of GAXML. Decidability for full verification holds for recursion-free GAXML. 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 GAXML systems.

**Acknowledgments** We wish to thank the participants in the ANR Docflow project for discussions on the verification of AXML systems, and in particular, Albert Benveniste, Eric Fabre, Blaise Genest, Loïc Hélouët, Anca Muscholl and Olivier Serre. We also thank Albert Benveniste and Pierre Bourrhis for discussions on the complete Mail Order example, that has been influential in the specification of our model.

## References


[1] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. *Proc.* ACM PODS 2004: 35-45.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. VLDB J. 17(5): 1019-1040 (2008).

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*, Addison-Wesley, 1995.

[4] Active XML homepage. http://activexml.net/

[5] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Proc.* ACM PODS 2005: 25-36.

[6] A. Benveniste and L. Helouet, Composition of GAXML services. Private communication, 2008.

[7] W. Fan and L. Libkin, On XML Integrity Constraints in the Presence of DTDs. *Proc.* ACM PODS 2001: 114-125.

[8] M. Arenas, W. Fan and L. Libkin, On Verifying Consistency of XML Specifications. *Proc.* ACM PODS 2002: 259-270.

[9] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. JCSS 66(4): 688-727 (2003). Also *Proc.* ACM PODS 2001: 138-149.

[10] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on words with data. *Proc.* IEEE LICS 2006: 7-16.

[11] E. Borger, E. Gradel and Y. Gurevich, *The Classical Decision Problem*, Springer 1997.

[12] C. David. Complexity of Data Tree Patterns over XML Documents, *Manuscript*.

[13] S. Demri and R. Lazic. LTL with the Freeze Quantifier and Register Automata. *Proc.* IEEE LICS 2006: 17-26.

[14] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. J. Comput. Syst. Sci. 73(3): 442-474 (2007). Also *Proc.* ACM PODS 2004.

[15] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A Verifier for Interactive, Data-Driven Web Applications. *Proc.* ACM SIGMOD 2005: 539-550.

[16] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. *Proc.* ACM PODS 2006: 90-99.

[17] E. Allen Emerson, *Temporal and Modal Logic*, in *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Sematics*, (ed. J. Van Leeuwen), North-Holland/MIT Press, 1990.

[18] B. Genest, A. Muscholl, O. Serre, M. Zeitoun. Tree pattern rewriting systems. *Proc.* Automated Technology for Verification and Analysis (ATVA) 2008: 332-346.

[19] R. Hull, M. Benedikt, V. Christophides and J. Su. E-Services: a look behind the curtain. *Proc.* ACM PODS 2003: 1-14.

[20] R. Khalaf, A. Keller, and F. Leymann, Business Processes for Web Services: Principles and Applications. IBM Systems Journal, Volume 45, Number 2, IBM Corp., 2006.

[21] M. Minsky. *Computation, Finite and Infinite Machines.* Prentice Hall, 1967.

[22] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. ACM Transactions on Computational Logic 15(3): 403-435 (2004).

[23] A. Nigam, N.S. Caswell. Business Artifacts: An approach to operational specification. IBM Systems Journal, 2003.

[24] L. Segoufin. Static Analysis of XML Processing with Data Values. In *Sigmod Record* 36(1), 2007.

[25] The Extensible Markup Language (XML) 1.0 (2nd Edition). http://www.w3.org/TR/REC-xml.

[26] The W3C Web Services Activity. http://www.w3.org/2002/ws.

## Appendix: The Mail Order Example

We provide here 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:

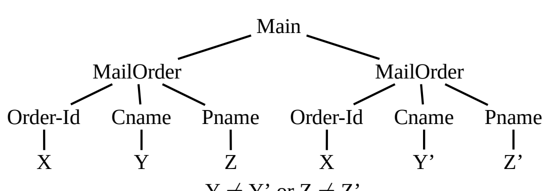
1. Receive an order from a customer `Cname` for a product `Pname`. The order is given a unique identifier `Order-ID` (uniqueness is enforced by the data constraint specified further).

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`.

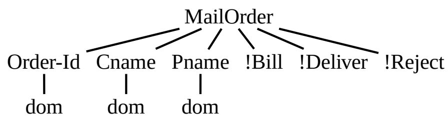
4. If the payment is correct (the catalog price of the product `Pname` is the paid `Amount`) then deliver the product by calling the external function `Deliver`. Otherwise reject the order by calling the external function `Reject`.

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 further. The uniqueness of mail order IDs is enforced by the data constraint consisting of the negation of the following pattern:

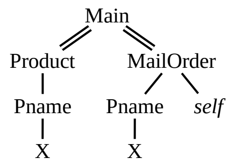



We next provide the specifications of functions.

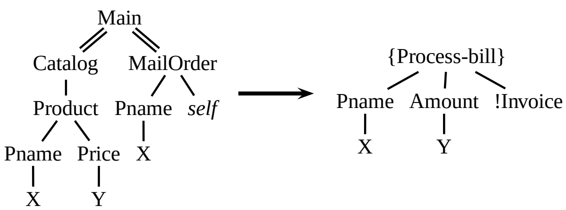
**MailOrder** is external and continuous. Its call guard is *true* and argument query empty. Its result has the following type, enforced by the DTD:



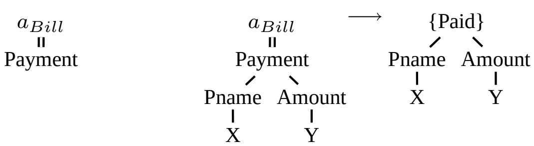

**Bill** is internal and non-continuous. Its call guard, that checks that the ordered product is available, is the following:




Its argument query is:

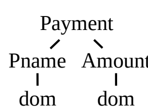



The return guard and query (also given in Example 2.2) are the following:

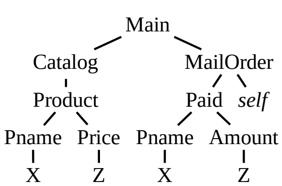



Return guard Return query

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

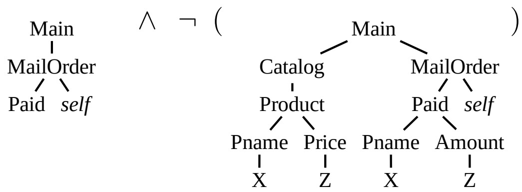



**Deliver** is external and non-continuous. Its call guard is




Its result consists of a single node labeled `Delivered` (this can be enforced by the DTD).

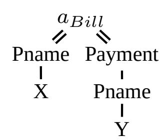
**Rejected** is external and non-continuous. Its call guard is the following:




Its result consists of a single node labeled `Rejected` (this can also be enforced by the DTD).

This completes the specification of the Mail Order GAXML 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:




$X \neq Y$