Static Analysis of Active XML Systems

Serge Abiteboul[*]
INRIA-Saclay & U. Paris Sud
France
www-rocq.inria.fr/~abitebou

Luc Segoufin
INRIA & LSV - ENS Cachan
France
www-rocq.inria.fr/~segoufin

Victor Vianu[†]
UC San Diego
USA
vianu@cs.ucsd.edu

ABSTRACT

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

Categories and Subject Descriptors: H.2.3 [Database Management]: Languages – XML

General Terms: Reliability, Theory, Verification

Keywords: Active XML, temporal logic, automatic verification

1. INTRODUCTION

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

Active XML documents [2, 4] (AXML for short) are XML documents [23] with embedded function calls realized as Web service calls [24]. In the spirit of [18, 21], a document is seen as a process that evolves in time. A function call is seen as a request to carry out a subtask whose result may lead to a change of state in the document. 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. 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.

[*]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.

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 total 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) was dealing with documents that do not evolve with time. Typically, they considered the consistency problem for XML specifications using DTDs and (foreign) key constraints [6, 7], the query containment problem [5] or the type checking problem [8]. This motivated studies of automata and logics on strings and trees over infinite alphabets [20, 12, 9]. See [22] for a survey on related issues.

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

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

Organization After presenting in Section 2 the GAXML model and the language Tree-LTL, we present in Section 3 the decidability and complexity results for recursion-free GAXML services. Relaxations of non-recursiveness are considered in Section 4, and shown to lead to undecidability. The decidability results on safety and bounded reachability are also presented in Section 4. The paper concludes with a brief discussion. Due to space limitations, most proofs are omitted or limited to informal sketches.

2. THE GAXML MODEL

We formalize in this section the GAXML model. To simplify the presentation, we consider a system with a single peer (we discuss this issue in Section 5). To illustrate our definitions, we use fragments of a Mail Order GAXML processing system, 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 n, m), *tags* Σ (denoted $a, b, c, \ldots$), *function names* $\mathcal{F}$, *data values* $\mathcal{D}$ (denoted $\alpha, \beta, \ldots$) *data variables* $\mathcal{V}$ (denoted $X, Y, Z, \ldots$), possibly with subscripts. The set $\mathcal{F}$ is the union of two disjoint sets of marked function symbols $\mathcal{F}^!$ and $\mathcal{F}^?$, where $\mathcal{F}^!$ is a set of symbols of the form $!f$, 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 and some result is expected (running call).

A *Guarded AXML* (GAXML) document is a tree whose internal nodes are labeled with tags in Σ and whose leaves are labeled by either tags, function names, 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 for controlling the activation of function calls and as a basis for our query language. A *pattern* is a forest of *tree patterns*. A *tree pattern* is a tree whose edges and nodes are labeled. An edge label indicates a child ($/$) or descendant ($//$) relationship. A node label either restricts the label of the node or is a variable denoting a data value. A constraint consisting of a Boolean combination of (in)equalities between the variables and/or data constants may also be given. Formally, a *tree pattern* is a tuple $(M, G, \lambda_M, \lambda_G)$, where:

- (M, G) is a 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 $\lambda_M(n) \in \Sigma \cup \{*\}$ for every internal node n,
- $\lambda_G : G \to \{/, //\}$.

Let P be a tree pattern and T a tree. A *matching* of P into T is a mapping μ 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) μ interprets $/$ as child and $//$ as descendant, (iii) μ preserves the label (with $*$ acting as a wildcard), (iv) nodes labeled with variables are mapped to data values.

A *pattern* 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 Q into a forest F is a mapping μ that is a matching of each P_i into some tree of F, and for which *cond* is satisfied. 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 Q holds in a forest F iff there exists at least one matching of Q into F. We then say that $Q(F)$ is true, otherwise it is false. This definition extends to Boolean combination of patterns by replacing each pattern Q by $Q(F)$. In particular this means that the patterns are matched independently of each other: If a variable X occurs in two different patterns Q and Q' of the

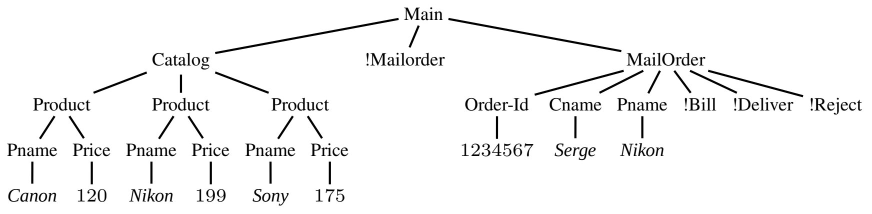


Figure 1: A GAXML document.

Boolean combination, then it is treated as quantified existentially in Q and independently quantified in Q'.

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, n) where F is a forest and n is a node of F. Such a pattern forces the node *self* in the pattern to be mapped to n. Figure 2 (b) provides an example of relative pattern. The pattern shown there checks that a product that has been ordered occurs in the catalog. It holds in the GAXML document of Figure 1 when evaluated at the unique node labeled `!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.

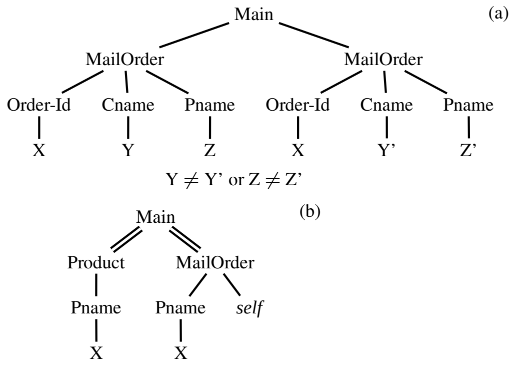


Figure 2: Two patterns

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

More formally, a *query* is an expression *Body* $\rightarrow$ *Head* where *Body* and *Head* are patterns such that for each H in *Head*,

- all its edges are labeled / (there are no descendant edges)
- its internal nodes have labels in Σ 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`.

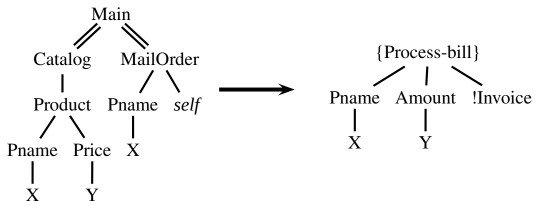


Figure 3: Example of a relative query

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

REMARK 2.1. *The constructor node provides explicit control over nesting of results. Note that this can be seen as syntactic sugaring in AXML, since the same effect can be achieved using function calls. However, the explicit constructor node is convenient from a specification viewpoint. Observe also that one could consider nesting of constructor nodes, in the spirit of* `group-by` *operators. Such an extension, which for simplicity we do not consider here, would not affect our results.*

Consider the evaluation of the query of Figure 3 on the 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* pattern of the query with X replaced by *Nikon* and Y by 199 (and no more 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 $a \in \Sigma$, the labels of children that a-nodes may have. As our trees are unordered we use Boolean combinations of statements of the form $|b| \geq k$ for $b \in \Sigma \cup \mathcal{F} \cup \{dom\}$

and k a non-negative integer[1]. (The word *dom* stands for any data value.) Validity of trees and of forests relative to a DTD is defined in the standard way. Details are omitted.

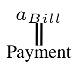
Schema and instance A GAXML *schema* S is a tuple $(\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ where

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

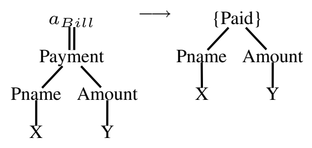
We next detail Φ_{int} and Φ_{ext}. For each $f \in \mathcal{F}$, let a_f be a new distinct label in Σ. Intuitively, a_f will be the label of the root of a tree where a call to f will be evaluated. (This tree may be seen as work space for the evaluation of the function.) Each function f of Φ_{int} is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

- $\mathit{arg}(f)$ (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.
- $\mathit{kind}(f) \in \{\mathit{non\text{-}continuous}, \mathit{continuous}\}$. 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.

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 query of `Bill` are:



Return guard



Return query

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

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

An instance of S is *valid* if it satisfies Δ.

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

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

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

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

Return of external call There exists some node n labeled $?f$ in some tree of $\mathcal{T}$, 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 n) is chosen arbitrarily. (Observe that constraints on the results of external calls can be imposed by Δ.)

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 query defining `!Bill` (see Figure 3). The dotted arrow indicates the function *eval*.

An *initial* instance of S is an instance of S consisting of a single tree whose root is not a function call and for which there is no running call.

An instance I is *blocking* if there is no instance I' such that $I \vdash I'$. A *run* of S is an infinite sequence $I_0, I_1, \ldots, I_i, \ldots$ of instances over S such that I_0 is an initial instance of S and for each $i \geq 0$, either $I_i \vdash I_{i+1}$ or I_i is blocking and $I_{i+1} = I_i$. Note that, for uniformity, we force all runs to be infinite by repeating a blocking instance forever if it is reached. A run is *valid* if all of its instances satisfy Δ. For a run ρ, we denote by $\mathit{adom}(\rho)$ the set of data values occurring in ρ, 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

[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.

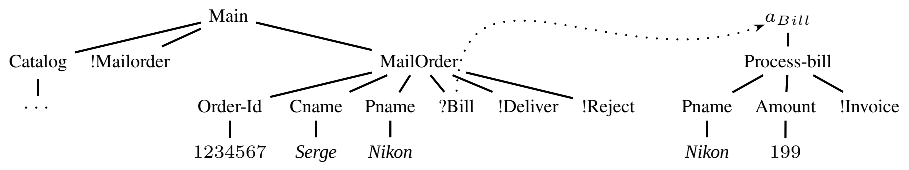


Figure 4: An instance with an $eval$ link

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 $\bar{X}$ some of its variables, designated as *free*. All other variables will be seen as quantified existentially, locally to P. (So logically, $P(\bar{X})$ may be seen as $\exists \bar{Y}(P)$, 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 [16]. Given a Tree-LTL formula φ, its free variables are the free variables of its patterns. A Tree-LTL sentence is an expression $\forall \bar{X} \varphi(\bar{X})$, where φ is a Tree-LTL formula and $\bar{X}$ are the free variables of φ. (As previously mentioned, variables that are not free are existentially quantified locally to each pattern.)

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

We now turn to the semantics of Tree-LTL. Intuitively, a sentence $\forall \bar{X}\varphi(\bar{X})$ 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 φ has no free variables. Consider a run ρ 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 ρ satisfies φ, denoted by $\rho \models \varphi$. Consider now a Tree-LTL sentence $\sigma = \forall \bar{X}\varphi(\bar{X})$. For a run ρ of S, we say that ρ satisfies $\forall \bar{X}\varphi(\bar{X})$, and denote this by $\rho \models \forall \bar{X}\varphi(\bar{X})$, if ρ satisfies $\varphi(h(\bar{X}))$ for each valuation h of $\bar{X}$ into *adom*(ρ). We say that S satisfies σ, denoted $S \models \sigma$, if every valid run of S satisfies σ.

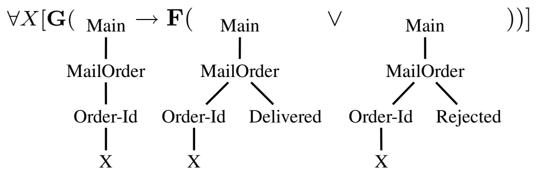
Two examples of Tree-LTL formulas are given below.
The branching-time variants Tree-CTL$^{(*)}$ are defined analogously. Not surprisingly, satisfaction of Tree-LTL sentences is undecidable for arbitrary GAXML systems. To obtain positive results, we need to place drastic but natural restrictions on these systems. We present in the next section such restrictions and 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 clearly are a source of difficulty for enforcing non-recursiveness syntactically, since an external function f may return some data with a call to some external function g, and g some data with a call to f. To circumvent this, we must assume some signature information on external functions. We do this by

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):

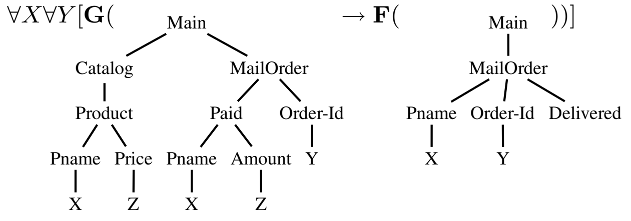


Figure 5: Some Tree-LTL formulas.

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 there is an edge from f to g if $g \in$ *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 Δ is non-recursive,

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

(iii) no function of S is continuous, and

(iv) the call graph of S is acyclic.

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

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

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

Next, 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 ξ iff its infinite extension satisfies ξ. We note the following useful fact. Its proof uses standard Büchi automata techniques, after replacing each pattern in ξ by a suitable proposition.

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

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 ξ.

PROPOSITION 3.4. *If there exists a valid pre-run of S satisfying ξ, then there exists a valid pre-run of the same length satisfying ξ, such that each of its instances has size doubly exponential in ξ 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 ξ. 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 ξ. The idea is to make sure that each R_m contains witnesses for all patterns in ξ satisfied by I_m, and also that it can mimic the transitions in the original run by keeping the "skeleton" of I_m (all paths from roots to nodes labeled with function symbols $?f$ or a_f) and also witnesses required to make the appropriate guards true. Satisfaction of the DTD must also be ensured, which requires additional witnesses. The construction is done in two passes: first, the needed witnesses are collected starting from I_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. □

We are now ready to show the desired upper bound. Let $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ be a recursion-free GAXML schema and φ a Tree-LTL sentence of the form $\forall \bar{X} \psi(\bar{X})$. Clearly, $S \not\models \varphi$ iff there is a valid run of S that satisfies $\neg\varphi = \exists \bar{X} \neg\psi(\bar{X})$. 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 a valid run ρ of S with domain $D \supseteq D_{\bar{X}}$ and a mapping h from $\bar{X}$ to $D_{\bar{X}}$ such that ρ satisfies $\xi = \neg\psi(h(\bar{X}))$ ($\psi(h(\bar{X}))$ is obtained from ψ by replacing, for each pattern in ψ for which $Y \in \bar{X}$ is a free variable, the label Y by $h(Y)$). In view of Propositions 3.2 - 3.4, a 2NEXPTIME decision procedure for checking whether $S \not\models \varphi$ is the following:

1. Guess $D_{\bar{X}}$ and the valuation h of $\bar{X}$ into $D_{\bar{X}}$; construct the formula ξ
2. Guess an initial instance R_0 of a valid pre-run of S, of size doubly exponential in S and ξ.
3. Generate non-deterministically a valid pre-run $R_0, \ldots, R_k$ of S; in the case of external function calls, guess an arbitrary answer of size at most doubly exponential in S and ξ. A blocking instance R_k is guaranteed to be reached after a number of transitions exponential in S.
4. Check that $R_0, \ldots, R_k$ satisfies ξ.

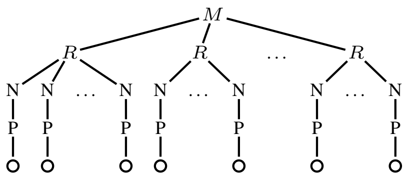
Note that (4) remains in 2NEXPTIME by Proposition 3.3.

We have established the following.

PROPOSITION 3.5. *It is decidable in CO-2NEXPTIME, given a recursion-free GAXML schema S and a Tree-LTL sentence φ, whether each valid run of S satisfies φ.*

Lower bound To show the matching lower bound, we consider a non-deterministic Turing Machine M running in time 2^{2^n} on inputs of size n. We construct a Tree-LTL sentence φ and a recursion-free GAXML schema S such that $S \not\models \varphi$ iff M accepts w. The main difficulty of the proof lies in simulating a Turing machine running in time 2^{2^n} with a recursion-free GAXML schema using only n functions and hence with runs of length 2^n. This requires a very efficient use of the available functions. We informally outline the construction.

The general idea is to ensure that $S \not\models \varphi$ iff some initial instance of S encodes an accepting computation of M on w. Initial instances of S have the following shape:



Each tree rooted at a symbol R is expected to code a configuration of M. Data values (not depicted above) are attached to the N and P-nodes in order to form a successor relation between the leaves (recall that our trees are unordered). The label of each leaf (the round circles in the figure above) is a tape symbol of M. In each R-subtree, the sequence of labels of the leaves induced by this successor relation codes a configuration of M. The global structure can easily be enforced using a DTD.

The difficult part of the construction is checking that the instance indeed holds a successor relation coded using the data values attached to each N-node and P-node. More precisely, define a directed graph G as follows. Its vertices are the data values of N-nodes. There is an edge (α, β) in G iff α and β are distinct data values of N-nodes x and y, such that the data value of the P-child of y is α. We specify S such that we can detect whether G induces a sequence long enough in each R-subtree for coding a configuration of M and such that G also induces a sufficiently long sequence of configurations. In particular, for coding the run of M, G must contain a sequence of length $2^{(2^n)} \cdot 2^{(2^n)} = 2^{(2^{n+1})}$.

We need to distinguish between three kinds of N-nodes, N_{beg}, N_{last}, and N_{inner}. We ensure that each R-subtree contains exactly one occurrence of N_{beg}, one occurrence of N_{last}, and that all the remaining N-nodes are N_{inner}. This can be done by adding a child to each N-node with a label identifying its kind. The constraints above can then be enforced using a DTD. We use these nodes to code respectively the first, last, and the other elements of the sequence induced by G on each R-tree. Similarly, we mark one of the R-subtree as the initial configuration, one as the final configuration, and denote them by R_{beg} and R_{last}. Again this can be enforced by a DTD.

It is easy to enforce, using data constraints, that every node of G has at most one outgoing edge and at most one incoming edge, and that G has no self loops. We can also make sure that the next element of a N_{last} node can only be a N_{beg} node. It remains to take care of loops and of sequences that may stop abruptly. For this we use function calls and compute, step by step, the transitive closure of G and the nodes at distance $2^{(2^{n+1})}$. Then suitable Tree-LTL formulas can check that G has the right format.

The transitive closure of G is computed by induction as follows: If T is the relation computed at some step, the next step computes $\exists z T(x,z) \wedge T(z,y)$. The double recursion allows to detect the cycles of G of length up to $2^{(2^{n+1})}$. This is enough to simulate M, because M terminates on w in at most $2^{(2^n)}$ steps.

With the successor relation in place, it remains to check that consecutive R-trees hold consecutive configurations of M. Checking this efficiently requires some additional non-trivial bookkeeping whose details we omit. In summary, the S and φ constructed from M and w are such that S violates φ iff M accepts w. This establishes the desired lower bound.

PROPOSITION 3.6. *It is* CO-2NEXPTIME-*hard to check whether a recursion-free GAXML schema satisfies a Tree-LTL sentence.*

We now have the main result of the section.

THEOREM 3.7. *It is* CO-2NEXPTIME-*complete to decide, given a recursion-free GAXML schema S and a Tree-LTL sentence φ, whether each valid run of S satisfies φ.*

REMARK 3.8. *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, for GAXML schemas whose call graph is a tree (a likely occurrence when functions model a hierarchical set of tasks) the complexity goes down to* CO-NEXPTIME. *Within the broader landscape of static analysis, this is quite reasonable. For instance, recall that even satisfiability of Barnays-Schönfinkel FO sentences, a much simpler question, already has complexity* NEXPTIME *[10].*

Using similar techniques, we can show decidability of other useful static analysis tasks for recursion-free GAXML.

THEOREM 3.9. *The following are decidable in* CO-2NEXPTIME *for a recursion-free GAXML schema $S = (\Phi_{int}, \Phi_{ext}, \Delta)$:*

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

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

For typechecking, the proof is analogous to that of Proposition 3.4. Suppose Δ consists of a DTD Δ' and a data constraint ψ. We first typecheck Δ': we show that whenever $\rho = I_0, \ldots, I_k$ is a prefix of a run of S such that I_0 satisfies Δ, I_k satisfies Δ'. Suppose, to the contrary, that there exists $\rho = I_0, \ldots, I_k$ such that I_0 is an initial instance (satisfying Δ), $I_m \vdash I_{m+1}$, and I_k violates Δ'. We construct a sequence $\rho' = R_0, \ldots, R_k$ with the same properties, such that the size of ρ' 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 Δ' can be done in 2NEXPTIME, so typechecking with respect to Δ' is in CO-2NEXPTIME. Now consider Δ. If the answer to the above is negative (there is a violation of Δ') then we are done (Δ 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 3.7. In summary, typechecking is decidable in CO-2NEXPTIME. □

REMARK 3.10. *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 Δ). A more lenient variant would typecheck subject to the* assumption *that results from calls to external functions do not cause violations. The decidability result of Theorem 3.9 can be easily extended to this variant.*

4. BEYOND RECURSION-FREE

In this section we prove that decidability of satisfaction of a Tree-LTL formula by a 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 Δ, (3) continuous functions, (4) a cyclic call graph.

For (1), undecidability is a simple consequence of the fact that satisfiability of Boolean combination of patterns in the presence of a DTD is already undecidable [11]. The first result concerns extensions (2-3). We prove a strong undecidability result, showing that even reachability of an instance satisfying a single positive pattern without variables becomes undecidable with any of these extensions. Furthermore, the result holds for schemas without data constraints and using no external functions. The proof is by reduction from the implication problem for functional and inclusion dependencies (FDs and IDs), known to be undecidable (see [3]).

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.*

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

THEOREM 4.2. *It is undecidable, given a positive pattern P without variables and a 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.*

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 φ, does every run starting from I satisfy φ? 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 sub-patterns. Specifically, let us allow labeling by $\neg$ one subtree of the pattern, with the safety restriction that all variables occurring in the negative subtree must also occur positively in the pattern. The semantics is the natural one: a match requires the positive part of the subtree to be matched to the input document, and the negative subtree to not be matched. An example of such query is: $r[/a/X][\neg\ /b/X]$. We show the following, using again 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.*

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 φ, whether S satisfies φ.*

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*[2] **AXEG** $(\neg P)$.

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 3.5.

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

Another practically significant problem is *bounded reachability*: for given k, is it possible to reach in at most k steps an instance satisfying a Boolean combination φ of patterns? The following is shown similarly to the proof of Theorem 3.7.

THEOREM 4.8. (**Bounded reachability**) *It is decidable in* 2NEXPTIME, *given a GAXML schema S, a Boolean combination φ 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, φ and k, is it the case that every instance of S reachable in at most k steps satisfies φ ? Clearly, this is the case iff no instance satisfying $\neg\varphi$ can be reached in at most k steps. Thus, bounded safety can be decided in CO-2NEXPTIME (and CO-NEXPTIME for fixed k).

5. DISCUSSION

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

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

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.

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

Acknowledgements We wish to thank the participants in the ANR Docflow project for discussions on the verification of AXML systems, and in particular, Albert Benveniste and Anca Muscholl.

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

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

[8] 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.

[9] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on words with data. In *LICS'06*, pp. 7-16, 2006.

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

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

[12] S. Demri and R. Lazic. LTL with the Freeze Quantifier and Register Automata. In *LICS'06*, pp. 17-26, IEEE 2006.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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).

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

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

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

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

APPENDIX

Running 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:

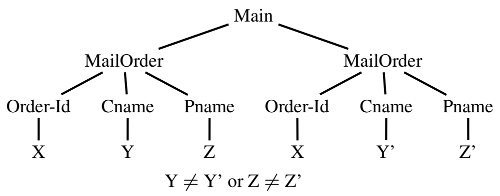
1. Receive an order from a customer `Cname` for a product `Pname`. The order is given a unique identifier `Order-ID` (uniqueness is enforced by the data constraint specified further).

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`.

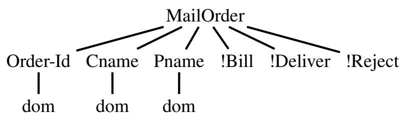
4. If the payment is correct (the catalog price of the product `Pname` is the paid `Amount`) then deliver the product by calling the external function `Deliver`. Otherwise reject the order by calling the external function `Reject`.

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:

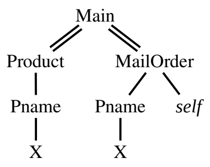


We next provide the specifications of functions.

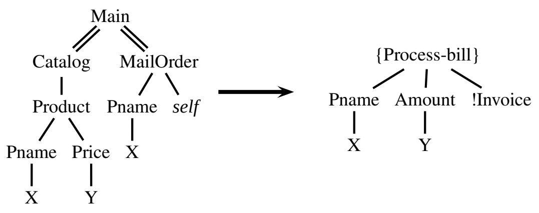
MailOrder is external and continuous. Its call guard is *true* and argument query empty. Its result has the following type, enforced by the DTD:



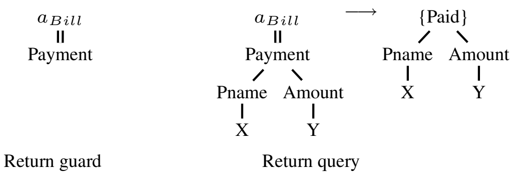
Bill is internal and non-continuous. Its call guard, that checks that the ordered product is available, is the following:



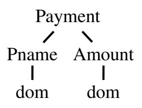
Its argument query is:



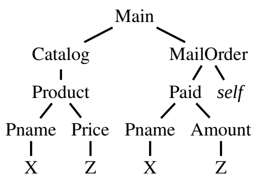
The return guard and query (also given in Example 2.2) are the following:



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):

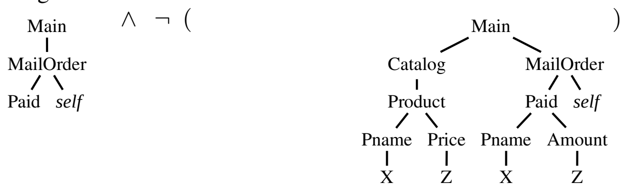


Deliver is external and non-continuous. Its call guard is



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

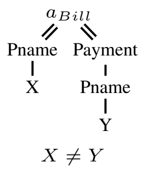
Rejected is external and non-continuous. Its call guard is the following:



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:


a_{Bill}
Pname
Payment
X
Pname
Y
$X \neq Y$