Comparing Workflow Specification Languages: A Matter of Views*

Serge Abiteboul **Pierre Bourhis** **Victor Vianu**†
INRIA Saclay, ENS Cachan & U. Paris Sud U.C. San Diego
Firstname.Lastname@inria.fr vianu@cs.ucsd.edu

ABSTRACT

We address the problem of comparing the expressiveness of workflow specification formalisms using a notion of *view* of a workflow. Views allow to compare widely different workflow systems by mapping them to a common representation capturing the observables relevant to the comparison. Using this framework, we compare the expressiveness of several workflow specification mechanisms, including automata, temporal constraints, and pre-and-post conditions, with XML and relational databases as underlying data models. One surprising result shows the considerable power of static constraints to simulate apparently much richer workflow control mechanisms.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Data Manipulation Languages; H.4.1 [**Information Systems Applications**]: Office Automation—*Workflow Management*

1. INTRODUCTION

There has recently been a proliferation of workflow specification languages, notably data-centric, in response to the need to support increasingly ubiquitous processes centered around databases. Prominent examples include e-commerce systems, enterprise business processes, health-care and scientific workflows. Comparing workflow specification languages is intrinsically difficult because of the diversity of formalisms and the lack of a standard yardstick for expressiveness. In this paper, we develop a flexible framework for comparing workflow specification languages, in which the pertinent aspects to be taken into account are defined by *views*. We use it to compare the expressiveness of several workflow specification mechanisms based on automata, pre/post conditions, and temporal constraints.

Consider a system that evolves in time as a result of internal computations or interactions with the rest of the world. Fundamentally, a workflow specification imposes constraints on this evolution. There are numerous approaches for specifying such constraints. Perhaps the most popular consists of specifying a set of abstract states of the system and imposing state transition constraints, in the spirit of a BPEL program [13]. Another, more declarative approach is to define a set of tasks equipped with pre/post conditions, such as IBM's Business Artifact model (see Related Work). Artifact systems may also impose constraints by temporal formulas on the history of the run ([26]).

The richness and variety of these approaches renders their comparison difficult. In particular, little is known of their relative expressive power. This is the main focus of the present paper.

We argue that a very useful approach for comparing workflow specification languages is provided by the notion of *workflow view*. More broadly, the notion of view is essential in the context of workflows, and the need to provide different views of workflows is pervasive. For example, views can be used to explain a workflow or provide customized interfaces for different classes of stakeholders, for convenience or privacy considerations. The interaction of workflows, and contractual obligations, are also conveniently specified by views. The design of complex workflows naturally proceeds by refinement of abstracted views. Views can be used at runtime for surveillance, error detection, diagnosis, or to capture continuous query subscriptions. The abstraction mechanism provided by views is also essential in static analysis and verification.

Depending on the specific needs, a workflow view might retain information about some abstract state of the system and its evolution, about some particular events and their sequencing, about the entire history of the system so far, or a combination of these and other aspects. Even if not made explicit, a view is often the starting point in the design of workflow specifications. This further motivates using views to bridge the gap between different specification languages. To see how this might be done, consider a workflow W specified by tasks and pre/post conditions and another workflow W' specified as a state-transition system, both per-

*This work has been partially funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant Webdam, agreement 226513. It also received partial support from the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.
†This author was supported in part by the NSF under award III-0916515.

ICDT 2011, March 21–23, 2011, Uppsala, Sweden.

taining to the same application. One way to render the two workflows comparable is to define a view of W as a state-transition system compatible with W'. This can be done by defining states using queries on the current instance and state transitions induced by the tasks. To make the comparison meaningful, the view of W should retain in states the information relevant to the semantics of the application, restructured to make it compatible with the representation used in W'. More generally, views may be used to map given workflows models to an entirely different model appropriate for the comparison. We will formalize the general notion of view and introduce a form of bisimulation over views to capture the fact that one workflow simulates another.

In our formal development, we mostly use the Active XML model [1], which provides seamless integration of complex data and processes. To describe system evolution (in the absence of workflow constraints), we use a core model called *Basic Active XML* (BAXML for short). BAXML documents are abstractions of XML with embedded service calls. A BAXML document is a forest of unordered, unranked trees, whose internal nodes are labeled with tags from a finite alphabet and whose leaves are labeled with tags, data values, or function symbols. The document evolves as a result of function calls that initiate new sub-tasks, and returns of results of function calls (using some local rewritings). The functions can be internal or external, the latter modeling interaction with the environment. For example, a BAXML document is shown in Figure 1. Documents are subject to static constraints specified by a DTD and a Boolean combination of tree-patterns. Note that this already provides some form of control on the execution flow, since a function call can be activated, or its result returned, only if the resulting instance does not violate the static constraints. Indeed, we will see that this already provides very powerful means to enforce workflow constraints.

BAXML provides a very natural framework for specifying runs of systems in which tasks correspond to evolving documents, and function calls are seen as requests to carry out sub-tasks. With the core model in place, we consider three ways of augmenting BAXML with explicit workflow control, corresponding to three important workflow specification paradigms:

Automata The automata are non-deterministic finite-state transition systems, in which states have associated tree pattern formulas with free variables acting as parameters. A transition into a state can only occur if its associated formula is true. In addition, the automaton may constrain the values of the parameters in consecutive states.

Guards These are pre-conditions controlling the firing of function calls and the return of their answers. This control mechanism was introduced in [5], where the results concern verification of temporal properties of such systems.

Temporal properties These are expressed in a temporal logic with tree patterns and Past LTL operators. A temporal formula constrains the next instance based on the history of the run.

Although presented here in the context of BAXML, these extensions capture the essential aspects of the three specification paradigms regardless of the specific underlying data model.

Our main results concern the relative power of BAXML and its extensions as workflow specification languages. When we insist that they generate *exactly* the same runs, the three extensions turn out to be incomparable. More interestingly, we then consider a more permissive and realistic notion of equivalence in which a view allows to hide portions of the data and some of the functions, thus providing more leeway in simulating one workflow by another. Surprisingly, we show that the core BAXML alone is largely capable to simulate the three specification mechanisms based on guards, automata, and temporal properties. This indicates the considerable power of static constraints to simulate apparently much richer workflow control mechanisms. Of course, specifications using guards, automata, and temporal properties are typically much more readable than their equivalent specifications in BAXML using hidden functions and static constraints.

The above results show the usefulness of seeing a workflow abstractly as a constraint on the runs of an underlying system, decoupled from the specific approach for defining the constraint. It also demonstrates the effectiveness of views in comparing workflows and worklow specification languages. Although the above languages are formalized in a specific Active XML context, we believe that the results demonstrate the wide applicability of the approach beyond this particular setting. In particular, the proofs provide general insight into when and how specifications based on automata, guards, and temporal constraints can simulate each other.

After settling the relative expressiveness of the languages using BAXML as a common core, we finally consider IBM's business artifact model, which uses a different paradigm based on the relational model and services equipped with first-order pre/post conditions. Relying once again on the views framework, we compare BAXML to the business artifact model, as formalized in [16]. We prove that BAXML can simulate artifacts, but the converse is false. The first result uses views mapping XML to relations and functions to services, so that artifacts become views of BAXML systems. For the negative result we use views retaining just the trace of function and service calls from the BAXML and the artifact system. This is a powerful result, since it extends to *any* views exposing *more* information than the function/service traces. The latter results demonstrate once again the flexibility and power of the views approach to comparing workflows.

Related work.

Workflow modeling and specification has traditionally been process centric (e.g., [22, 39]). This has been captured in the workflows community by flowcharts, Petri nets [40, 41, 7], and state charts [25, 32]. The comparison of such systems using the notion of bisimulation is considered in [31, 38]. More recently, data-centric workflows have been considered in [42], and in particular the *artifact* model of IBM [35]. Verification for such models is considered in [23, 24, 11, 16, 21]. The comparison of such systems is considered in [14] using the notion of dominance, which focuses on the input/output pairs of the workflows. Other models in the same spirit include the Vortex workflow framework [28, 18, 27], the OWL-S proposal [30, 29] as well as some work on semantic Web services [33]. The article [17] (building on [37, 6]), considers the verification of properties of data-centric workflows specified in LTL-FO, first-order logic extended with linear-

time temporal logic operators. Similar extensions have been previously used in various contexts [19, 3, 37]. Apart from the work on verification of BAXML with guards mentioned above [5], most other work on static analysis on XML (with data values) deals with documents that do not evolve in time, e.g., [20, 10, 8]. This motivated studies of automata and logics on strings and trees over infinite alphabets [34, 15, 12]. See [36] for a survey on related issues.

A survey on Active XML may be found in [1]. In [2], active XML documents are used to capture data and workflow management activities in distributed settings, in the spirit of the artifact approach. The study of the interplay between queries and sequencing in the artifact approach was the driving motivation of the present work.

Organization.

The paper is organized as follows. We introduce the view-based framework for comparing workflow languages in Section 2. The BAXML model and the workflow languages are presented in Sections 3 and 4. Their expressive power with respect to different views is compared in Section 5. In Section 6 we compare BAXML with a variant of IBM's business artifacts, and show that BAXML can simulate artifacts, but the converse is false. We end with brief conclusions. Due to space limitations, most proofs are omitted.

2. VIEWS AND SIMULATIONS

In this section, we introduce an abstract framework for workflows and views of workflows. We then use it to compare workflows.

Workflow Systems and Languages

The model for workflows we consider is quite general. Intuitively, a workflow system describes the tree of the possible runs of a particular system. More formally, the nodes of a workflow system are labeled by *states* from an infinite set Q_∞ and the edges by *events* from an infinite set E_∞ ($Q_\infty \cap E_\infty = \emptyset$). For example, a state of a workflow system may be an instance of a relational database or an XML document. It may also include various other relevant information such as the state of an automaton controlling the workflow, or historical information such as the prefix of the run leading up to it. A typical event may consist of the activation of a task, including its parameters. The presence of data explains why the sets Q_∞ and E_∞ are taken to be infinite.

The workflow systems we consider include two particular events, namely *block* and ϵ, both in E_∞, whose role we explain briefly. First consider *block*. For uniformity, it is convenient to assume that all runs are infinite. To this end, we use the distinguished event *block* to signal that the system has reached a terminal state that repeats forever (so once a system blocks, it remains blocked).

On the other hand, the ϵ event corresponds to the classical notion of *silent transition*. Its meaning is best explained in the context of a view (to be formally defined further), which defines the observable portion of states and events. In particular, it may hide information about states as well as events in the source system. For a transition in the source system, if the event is (even partially) visible in the view or if the state of the view changes, the transition is observable in the view. On the other hand, it may be the case that both the event and the state change are invisible in the view. So, although there has been a transition in the workflow system, nothing can be observed in the view. This is modeled by a silent transition, indicated by the special event ϵ. Observe that, unlike for blocking transitions, an ϵ transition may be followed in the view by non-ϵ (visible) transitions, in which the state may change.

More formally:

DEFINITION 1 (WORKFLOW SYSTEM). *A workflow system is a tuple* $(N, n_0, \delta, q_0, \lambda_N, \lambda_\delta)$ *where:*

- (N, n_0, δ) *is a tree with root* n_0*, nodes* N*, edges* δ*.*
- *all maximal paths from* n_0 *are infinite.*
- λ_N *is a function from* N *to* Q_∞*, and* $\lambda_N(n_0) = q_0$*.*
- λ_δ *is a function from* δ *to* E_∞*.*
- *for each* $(n, n') \in \delta$*, if* $\lambda_\delta((n, n')) = \epsilon$ *then* $\lambda_N(n) = \lambda_N(n')$*.*
- *for each* $(n, n') \in \delta$*, if* $\lambda_\delta((n, n'))$ = block *then* n' *is the only child of* n *and* $\lambda_N(n) = \lambda_N(n')$*. Moreover,* n' *has only one outgoing edge also labeled* block*.*

The edges in δ are also called *transitions* of the workflow, and q_0 is called its *initial state*.

Finally, a *workflow language* $\mathcal{W}$ consists of an infinite set of expressions, called workflow specifications. For example, BAXML, and its extensions with guards, automata, and temporal constraints, are all workflow languages. Given a workflow language $\mathcal{W}$ and $W \in \mathcal{W}$, the semantics of W is a workflow system (i.e., the tree of runs defined by W) and is denoted by $[W]_\mathcal{W}$, or $[W]$ when $\mathcal{W}$ is understood.

Views of Workflow Systems

We next formalize the notion of view of a workflow system. We will argue that this is an essential unifying tool for understanding diverse workflow models. In the present paper, we rely heavily on the notion of view in order to compare workflows languages.

A *view* V is a mapping on $Q_\infty \cup E_\infty$, such that $V(Q_\infty) \subseteq Q_\infty$, $V(E_\infty) \subseteq E_\infty$, $V(\epsilon) = \epsilon$, and $V(e) = \mathit{block}$ iff $e = \mathit{block}$. This mapping is extended to workflow systems as follows. Let $WS = (N, n_0, \delta, q_0, \lambda_N, \lambda_\delta)$ and V be a view. Then $V(WS)$ is defined[1] as $(N, n_0, \delta, V(q_0), \lambda_N \circ V, \lambda_\delta \circ V)$. We say that the view V is *well-defined for* WS if $V(WS)$ is a workflow system.

Note that, by definition of the mapping, the properties of blocking transitions are automatically preserved. Note also that, by definition of well-defined workflow system, for each $(n, n') \in \delta$, if $V(\lambda_\delta((n, n'))) = \epsilon$ then $V(\lambda_N(n)) = V(\lambda_N(n'))$.

Simulation of Workflows

We next consider the comparison of workflow systems and workflow languages based on the concept of view. We use a variant of bisimulation [31] (that we call w-bisimulation). Of course, many other semantics for comparison are possible. We refrain from attempting a taxonomy of such semantics, and instead settle on one definition that is quite general and adequate for our purposes.

[1]Composition is applied left-to-right.

In our semantics, we wish to be able to capture silent transitions as well as infinite branches of such transitions. Given a workflow system as above, for each $e \in E - \{\epsilon\}$, we define the relation $\stackrel{e}{\rightarrow}$ on nodes by $n \stackrel{e}{\rightarrow} m$ if there is a sequence of transitions from n to m, all of which are silent except for the last one, which is labeled e.

Informally, the silent transitions are seen as partial internal computation that do not have impact for the possible observable reachable events. The choices made during the internal computation may be different, but the visible transitions at the end of sequences of silent transitions are the same.

DEFINITION 2 (W-BISIMULATION). *Let*

$$\mathrm{WS}_i = (N^i, n_0^i, \delta^i, q_0, \lambda_N^i, \lambda_\delta^i)$$

$i \in \{1, 2\}$, be two workflow systems (with the same initial state). A relation B from N^1 to N^2 is a w-bisimulation *of* WS_1 *and* WS_2 *if $B(n_0^1, n_0^2)$ and for each n_1, n_2 such that $B(n_1, n_2)$ the following hold:*

- $\lambda_N^1(n_1) = \lambda_N^2(n_2)$.
- *For each event $e \neq \epsilon$, if $n_1 \stackrel{e}{\rightarrow} n_1'$ in* WS_1 *then there exists n_2' such that $n_2 \stackrel{e}{\rightarrow} n_2'$ in* WS_2 *and $B(n_1', n_2')$, and conversely.*
- *there is an infinite path of silent transitions from n_1 in* WS_1 *iff there is an infinite path of silent transitions from n_2 in* WS_2.

We denote by $\mathrm{WS}_1 \sim \mathrm{WS}_2$ *the fact that there exists a w-bisimulation of* WS_1 *and* WS_2.

We note that there are well-known notions of bisimulation related to ours, such as weak-bisimulation and observation-congruence equivalence, motivated by distributed algebra [31]. These differ from w-bisimulation in their treatment of silent transitions. For example, infinite paths of silent transitions are relevant to w-simulation but are ignored in weak bisimulation. It can be seen that observation-congruence equivalence implies w-bisimulation, but weak bisimulation and w-bisimulation are incomparable.

Clearly, $\sim$ is an equivalence relation. Observe that views preserve w-bisimulation. More precisely, let $WS_1 \sim WS_2$. Then for each view V,

(*) $V(WS_1)$ is well-defined iff $V(WS_2)$ is well-defined, in which case $V(WS_1) \sim V(WS_2)$.

Equivalence of workflow systems as previously defined essentially requires the two systems to have the same set of states and events. However, in general we wish to compare workflow systems whose states and events may be very different. In order to make them comparable, we use *views* mapping the states and events of each system to a common, possibly new set of states and events. Intuitively, these represent abstractions extracting the observable information relevant to the comparison. The views may also involve substantial restructuring, thus extending classical database views.

Suppose we wish to compare languages $\mathcal{W}_1$ and $\mathcal{W}_2$. To compare workflow specifications in $\mathcal{W}_1$ and $\mathcal{W}_2$, we use sets of views $\mathcal{V}_1$ and $\mathcal{V}_2$ that map the states and events of $\mathcal{W}_1$ and $\mathcal{W}_2$ to a common set.

DEFINITION 3 (SIMULATION). *Let $\mathcal{W}_1, \mathcal{W}_2$ be workflow languages and $\mathcal{V}_1, \mathcal{V}_2$ be sets of views. The language $\mathcal{W}_2$* simulates *$\mathcal{W}_1$ with respect to $(\mathcal{V}_1, \mathcal{V}_2)$, denoted $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_2)} \mathcal{W}_2$, if for each $W_1 \in \mathcal{W}_1$ and $V_1 \in \mathcal{V}_1$ such that $V_1(W_1)$ is well-defined, there exist $W_2 \in \mathcal{W}_2$ and $V_2 \in \mathcal{V}_2$ such $V_2(W_2)$ is well-defined and $V_1(W_1) \sim V_2(W_2)$.*

REMARK 1. *Note that the definition of simulation does not require effective construction of the simulating workflow specification. However, all our positive simulation results are constructive. The negative result in Theorem 11 also concerns effective simulation.*

For sets of views $\mathcal{V}, \mathcal{V}'$, we define $\mathcal{V} \circ \mathcal{V}' = \{V \circ V' \mid V \in \mathcal{V}, V' \in \mathcal{V}'\}$. Intuitively, a view $V \circ V'$ is coarser than V (or equivalently, V is more refined than $V \circ V'$).

The following key lemma is a straightforward consequence of (*). It states that the relation $\hookrightarrow$ is stable under composition of views.

LEMMA 1 (COMPOSITION). *Let $\mathcal{W}_1$ and $\mathcal{W}_2$ be workflow languages and $\mathcal{V}_1, \mathcal{V}_2$ and $\mathcal{V}$ be sets of views. If $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_2)} \mathcal{W}_2$ then $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1 \circ \mathcal{V}, \mathcal{V}_2 \circ \mathcal{V})} \mathcal{W}_2$.*

The Composition Lemma allows to relate simulations relative to different classes of views. It says that simulation relative to given views implies simulation relative to any coarser views. This provides a tool for proving both positive and negative simulation results.

A useful version of the above lemma is the following, combining composition and transitivity.

LEMMA 2. *Let $\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3$ be workflow languages, and $\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3$ and $\mathcal{V}$ be sets of views. If $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_2 \circ \mathcal{V})} \mathcal{W}_2$ and $\mathcal{W}_2 \hookrightarrow_{(\mathcal{V}_2, \mathcal{V}_3)} \mathcal{W}_3$, then $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_3 \circ \mathcal{V})} \mathcal{W}_3$.*

As we will see, the version of transitivity provided by the above is routinely used in proofs that combine multiple stages of simulation.

3. THE BASIC AXML MODEL

In this section we present BAXML, the *Basic AXML model*. This is essentially a simplified version of the GAXML model of [5], obtained by stripping it of the control provided by call and return guards of functions (all such guards are set to *true*). We consider such control later as one of the workflow specification mechanisms. The section may be skipped by readers familiar with the GAXML model. Due to space constraints, and to limit duplication, the description is informal.

To illustrate our definitions, we use a simplified version of the Mail Order example of [5]. The purpose of the Mail Order system is to fetch and process individual mail orders. The system accesses a catalog subtree providing the price for each product. Each order follows a simple workflow whereby a customer is first billed, a payment is received and, if the payment is in the right amount, the ordered product is delivered.

In the model, trees are unranked and unordered. 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.

For each function name f, we also use the symbols $!f$ and $?f$, called *function symbols*, and denote by $\mathcal{F}^!$ the set $\{!f \mid f \in \mathcal{F}\}$ and by $\mathcal{F}^?$ the set $\{?f \mid f \in \mathcal{F}\}$. 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). After the answer of a call at node x is returned, the call may be kept or the node x may be deleted. If calls to $!f$ are kept, f is called *continuous*, otherwise it is *non-continuous*. For example, the role of the `MailOrder` function in Figure 1 is to indefinitely fetch new mail orders from customers, so `MailOrder` is specified to be continuous. On the other hand, the function `!Bill` occurring in a `MailOrder` is meant to be called only once, in order to carry out the billing task. Once the task is finished, the call can be removed. Therefore, `Bill` is specified to be non-continuous.

A BAXML document is a tree whose internal nodes are labeled with tags in Σ and whose leaves are labeled by either tags, function symbols, or data values. A BAXML forest is a set of BAXML trees. An example of BAXML document is given in Figure 1.

To avoid repetitions of isomorphic sibling subtrees, we define the notion of reduced tree. A tree is *reduced* if it contains no distinct isomorphic sibling subtrees without running calls $?f$. We henceforth assume that all trees considered are reduced, unless stated otherwise. However, note that the forest of an instance may generally contain multiple isomorphic trees.

Patterns. We use patterns as the basis for our query language, and later in the specification of workflow constraints and temporal properties. A *pattern* is a forest of *tree-patterns*. A *tree-pattern* is a tree whose edges are labeled by child (/) or descendant (//), where, as in XPath, descendant is reflexive. Nodes are labeled by tags if they are internal, and by tags, function symbols, or variables if they are leafs. In addition, nodes may be labeled by wildcard (*), which can map to any tag. A constraint consisting of a Boolean combination of (in)equalities between the variables and/or data constants may also be given. In particular, we can specify joins (equality of data values). A tree-pattern is evaluated over a tree in the straightforward way. The definition of the evaluation of patterns over forests extends the above in the natural way. 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 BAXML document of Figure 1. (Indeed, we want `Order-Id` to be a key).

We sometimes 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 BAXML 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. If a variable X occurs in two different patterns P and P' of the Boolean combination then it is treated as quantified existentially for P and independently quantified for P'.

It will be useful to occasionally consider *parameterized* patterns, in which some variables are designated as *free*. Let $P(\bar{X})$ be a pattern with free variables $\bar{X}$, and ν an assignment of data values to $\bar{X}$. A BAXML forest I satisfies $P(\bar{X})$ for assignment ν, denoted $I, \nu \models P(\bar{X})$, if I satisfies the pattern $P(\nu(\bar{X}))$ obtained by replacing each variable in $\bar{X}$ by its value under ν.

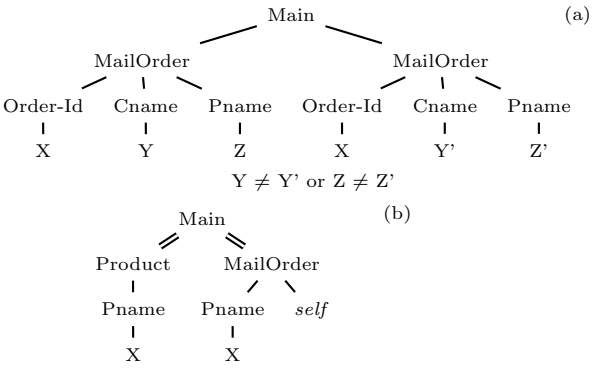


Figure 2: Two patterns

Queries. As previously mentioned, patterns are used in queries, as shown next. A *query* is a finite union of rules of the form $Body \rightarrow Head$, where *Body* and *Head* are patterns and *Head* contains no descendant edges and no constants, and all its variables occur in *Body*. In each tree of *Head*, all variables occur under a designated *constructor node*, specifying a form of nesting. When evaluated on a forest, the matchings of *Body* define a set of valuations of the variables. The answer for the rule is obtained by replacing, in each tree of *Head*, the subtree rooted at the constructor node with the forest obtained by instantiating the variables in the subtree with all their matchings provided by the *Body*. The answer to the query is the union of the answers for each rule. As for patterns, we may consider queries evaluated relative to a specified node in the input tree. A *relative query* is defined like a query, except that the bodies of its rules are relative patterns (P, *self*). An example of relative query (with a single rule) is given in Figure 3. The label of the constructor node (indicated by brackets) is `Process-bill`.

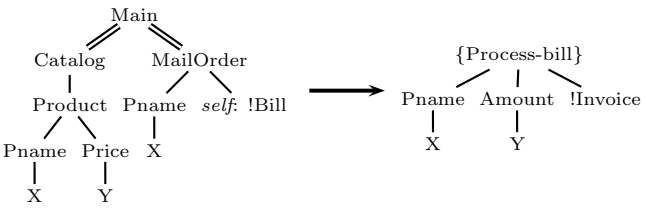


Figure 3: Example of a relative query

Consider the evaluation of the query of Figure 3 on the BAXML document of Figure 1 at the unique node labeled `!Bill`. There is a unique matching of the *Body* pattern and the result is the *Head* pattern of the query with X replaced by *Nikon* and Y by 199 (without brackets for `Process-bill`).

DTD. Trees used by a BAXML system may be constrained using DTDs and Boolean combinations of patterns. For DTDs, we use a typing mechanism that restricts, for each

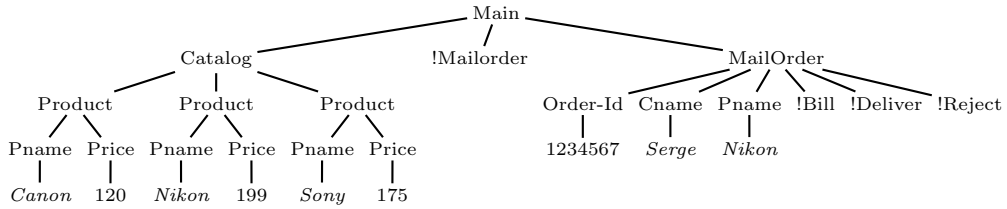


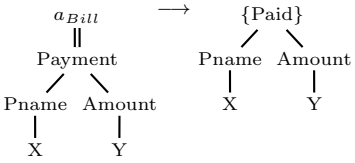
Figure 1: A BAXML document.

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 $b \in \Sigma \cup \mathcal{F}^! \cup \mathcal{F}^? \cup \{dom\}$ and k a non-negative integer. Validity of trees and of forests relative to a DTD is defined in the standard way.

Schemas and instances. A BAXML *schema* S is a tuple $(\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ where (i) the set Φ_{int} contains a finite set of internal function specifications, (ii) the set Φ_{ext} contains a finite set of external function specifications, and (iii) Δ provides static constraints on instances of the schema. It consists of a DTD and a Boolean combination of patterns. For instance, the negation of the pattern in Figure 2 (b) states that `Order-Id` uniquely determines the mail order.

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 root of a subtree where a call to f is being evaluated. (This subtree may be seen as a task initiated by the function call.) The specification of a function f of Φ_{int} indicates whether f is continuous or not, provides its *argument query* (a relative query), and *return query* (a query rooted at a_f). When the argument query is evaluated, *self* binds to the node at which the call $!f$ is made. The role of the argument query is to define the argument of a call to f, which is also the initial state of the task corresponding to f.

Example 1 We continue with our running example. The function `Bill` used in Figure 1 is specified as follows. It is internal and non-continuous. 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 query of `Bill` is:



Each function f in Φ_{ext} is specified similarly, except that the return query is missing. In addition, a DTD Δ_f constrains the answers returned by f (the DTD assumes a virtual root under which the answer forest is placed). Intuitively, an external call can return any answer satisfying Δ_f at any time, as long as the resulting instance also satisfies the global static constraints Δ. For example, `MailOrder` is external, since its role is to fetch orders from an external user.

An *instance* I over a BAXML schema $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ is a pair $(\mathcal{T}, eval)$, where $\mathcal{T}$ is a BAXML 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: (i) for each n with label $?f$, $eval(n)$ is a tree in $\mathcal{T}$ with root label a_f (its workspace), and (ii) every tree in $\mathcal{T}$ with root label a_f is $eval(n)$ for some n labeled $?f$. An instance of S is *valid* if it satisfies Δ.

Runs. Let $I = (\mathcal{T}, eval)$ and $I' = (\mathcal{T}', eval')$ be instances of a BAXML schema $S = (\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$. The instance I' is a *possible next instance of* I iff I' is obtained from I by making a function call or by receiving the answer to an existing call. We refer to the latter as an *event*. More precisely, an event is an expression of the form $!f(F)$ or $?f(F)$, where f is a function, and F is the forest consisting of the argument, resp. answer to the function call. For technical reasons, we also use two special events, *init* that only generates the initial instance, and *block*, whose use will be clear shortly. We denote by $I \vdash_e I'$ the fact that I' is a possible next instance of I caused by event e.

We now provide more details. When a call to $!f$ is made at node n, the label of n is changed to $?f$. If f is internal, we additionally 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 of f on input $(\mathcal{T}, n)$. When an answer to call $?f$ at node n is received, the trees in the answer are added as siblings of n, and n is deleted (if f is non-continuous) or its label is reset to $!f$ (if f is continuous). If f is external, its answer is a forest satisfying Δ_f. If f is internal, the answer can be returned only if $eval(n)$ contains no running calls $?g$, in which case the answer consists of the result of evaluating the return query of f on $eval(n)$, after which $(n, eval(n))$ is removed from the graph of *eval*.

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 1. 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*.

We will typically be interested in *runs* of such systems. An *initial* instance of schema 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. For runs, we use a variation of the model of [5]. A *prerun* of a schema S is a finite sequence $\{(I_i, e_i)\}_{0 \leq i \leq n}$, such that (i) for each i, I_i satisfies the static constraints Δ, (ii) $e_0 = init$, and (iii) for each $i > 0$, $I_{i-1} \vdash_{e_i} I_i$. A *run* is an infinite sequence $\rho = \{(I_i, e_i)\}_{i \geq 0}$ such that:

nonblocking each finite prefix of ρ is a prerun of S, or

blocking there is a finite prefix $(I_0, e_0), ..., (I_n, e_n)$ of ρ that is a maximal prerun[2] of S; and for each $i > n$, $I_i = I_n$ and $e_i = block$.

[2]There is no (I', e') for which $(I_0, e_0), ..., (I_n, e_n)(I', e')$ is a prerun of S.

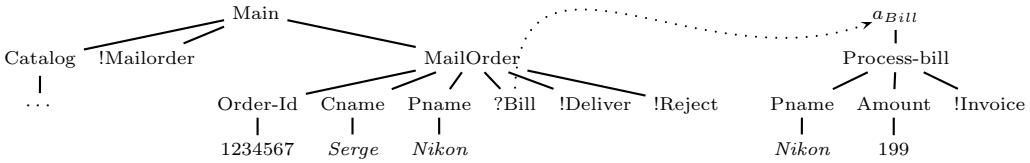


Figure 4: An instance with an *eval* link

Thus, we force all runs to be infinite by repeating forever a blocking instance from which no legal transition is possible, if such an instance is reached.

Semantics with and without aborts

We next discuss a subtle difference between the semantics adopted here and that of [5]. According to our semantics, if a prerun reaches an instance from which every transition leads to a violation of the static constraints, the prerun *blocks* forever in that instance, generating a blocking run. In contrast, the semantics of [5] allows blocking runs only if no transition exists at all (whether leading to a valid instance or not). If there are possible transitions but they all lead to constraint violations, the prerun is discarded. Intuitively, this amounts to aborting the run. We refer to this as the semantics of runs *with aborts*, and to the one we follow in this paper as the semantics of runs (*without aborts*). Note that in our semantics, every prerun is extensible to a (possibly blocking) run, whereas this is not the case in the semantics with aborts. Furthermore, as shown next, in the semantics with aborts it is undecidable if a given prerun can be extended to an infinite run. This is a main motivation for our choice of the semantics without aborts.

THEOREM 2. *Let S be a BAXML schema and ρ a prerun of S. Under the semantics with aborts, it is undecidable whether ρ is the prefix of a run of S. Furthermore, this remains undecidable even for nonrecursive*[3] *DTDs.*

The proof for arbitrary DTDs is trivial by the undecidability of satisfiability of static constraints. The proof for nonrecursive DTDs is by reduction of the implication problem for functional and inclusion dependencies (FDs and IDs), known to be undecidable (see [4]).

4. WORKFLOW CONSTRAINTS

In this section, we introduce three ways of enriching the BAXML model with workflow constraints: (i) function call and return guards (yielding the GAXML model), (ii) an automaton model, and (iii) temporal constraints. Each corresponds to a very natural way of expressing constraints on the evolution of a system. We study and compare these mechanisms in the next sections.

We begin by considering an abstract notion of workflow constraint. A *workflow constraint* W over a BAXML schema S is a prefix-closed property of preruns of S. For a prerun ρ of S, we denote by $\rho \models W$ the fact that ρ satisfies W. We denote by $S|W$ the workflow specification defined by S constrained by W. A *run* of $S|W$ is an infinite sequence $\rho = \{(I_i, e_i)\}_{i\geq 0}$ such that:

nonblocking each finite prefix of ρ is a prerun of S *that satisfies* W.

blocking there is a finite prefix $(I_0, e_0), ..., (I_n, e_n)$ of ρ that is a maximal prerun of S *satisfying* W; and for each $i > n$, $I_i = I_n$ and $e_i = block$.

Observe that nonblocking runs of $S|W$ are particular nonblocking runs of S. Also, a sequence $\{(I_i, e_i)\}_{i\geq 0}$ may be a blocking run of $S|W$ but not a blocking run of S. (This is because all transitions that are possible according to S are forbiddent by W.) The set of runs of $S|W$ is denoted by $runs(S|W)$.

A main goal of the paper is to compare the descriptive power of different formalisms for specifying workflow constraints. To this end, we consider the workflow languages $\mathcal{G}$ (for call guards), $\mathcal{A}$ (for automata), and $\mathcal{T}$ (for temporal formulas), defined next.

Call and return guards.

Recall the Mail Order example, in which processing an order requires executing some tasks in a desired sequence (order, bill, pay, deliver). Since tasks in BAXML are initiated by function calls, one convenient workflow specification mechanism is to attach guards to function calls. For instance, the guard of `!Deliver`, shown in Figure 5, might require that the ordered product must have been paid in the correct amount. Similarly, it is useful to control when the answer of an internal function may be returned. This can be done by providing *return guards*.

Let S be a BAXML schema. A *guard assignment* over S is a pair $\gamma = (\gamma_c, \gamma_r)$, where:

- γ_c, the *call guard* assignment, is a mapping from the functions of S to Boolean combinations of relative patterns over S. A call to f can only be activated if $\gamma_c(f)$ holds.

- γ_r, the *return guard* assignment, is a mapping from the functions of S which is *true* for external functions and a Boolean combination of tree patterns rooted at a_f for each internal function f. The result of a call to f is returned only when $\gamma_r(f)$ guard is satisfied on its current workspace. Return guards constrain only internal functions.

A prerun $\rho = (I_0, e_0), ..., (I_n, e_n)$ of S *satisfies* $\gamma = (\gamma_c, \gamma_r)$, denoted $\rho \models \gamma$, if for each transition $I_{i-1} \vdash_{e_i} I_i$, if the transition results from a function call to $!f$ at node u the guard $\gamma_c(f)$ holds in (I_{i-1}, u), and if the transition results from the return of an internal function call $?f$ at node u, $\gamma_r(f)$ holds in $eval_{i-1}(u)$. Observe that these constraints involve consecutive instances only.

The set of all guard workflow constraints is denoted $\mathcal{G}$. A *GAXML schema* is an expression $S|\gamma$, for some $\gamma \in \mathcal{G}$.

[3]A DTD is *recursive* is there is a cycle in the graph that has an edge from tag a to b if the DTD allows b to label a child of a node labeled a.

Example 3 Figure 5 shows call guards for some functions in the Mail Order example. The call guard of function `Bill` is given in Figure 2(b) (this checks that the ordered product is available). The call guard of `Invoice` is *true*. In the same

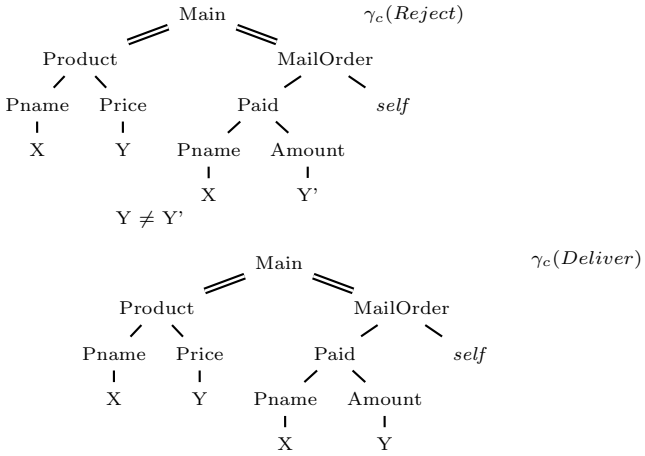
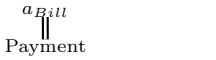


Figure 5: Call guards of *Reject* and *Deliver*.

example, the return guard of function `Bill` is:



indicating that payment has been received, so billing is completed.

Pattern automata. We next consider workflows based on automata. The states of the automaton are defined using pattern queries. The automaton has no final states, since BAXML (like AXML) does not have a built-in notion of successful computation.

A *pattern automaton* is a tuple $(Q, q_{init}, \delta, \Upsilon)$ where:

- Q is a finite set of states, $q_{init} \in Q$, and each $q \in Q$ has an associated set of variables $\overline{X}_q$;
- For each q, $\Upsilon(q)$ is a Boolean combination of parameterized patterns whose set of free variables equals $\overline{X}_q$;
- the transition function δ is a partial function over $Q \times Q$; for each q, q', $\delta(q, q')$ is a Boolean combination of equalities of variables in $\overline{X}_q$ and $\overline{X}_{q'}$.

To simplify the presentation, we assume without loss of generality that $\overline{X}_q$ and $\overline{X}_{q'}$ have no variables in common.

Let $\mathcal{A}$ be the set of pattern automata. An *AAXML schema* is an expression $S|A$ for a BAXML schema S and $A \in \mathcal{A}$. A prerun $\rho = \{(I_i, e_i)\}_{i \leq n}$ of S satisfies an automaton constraint A, denoted $\rho \models A$, if there exists a sequence $\{(q_i, \nu_i)\}_{i \leq n}$, where $q_0 = q_{init}$ and ν_i is a valuation of X_{q_i}, such that for each $i \leq n$:

1. $I_i, \nu_i \models \Upsilon(q_i)$,
2. $\nu_i(\overline{X}_{q_i}) \cup \nu_{i+1}(\overline{X}_{q_{i+1}}) \models \delta(q_i, q_{i+1})$.

Intuitively, the state of such an automaton after reading a finite sequence ρ of instances is a pair (q, ν) where ν is a valuation of the variables in $\overline{X}_q$. Note that the automaton

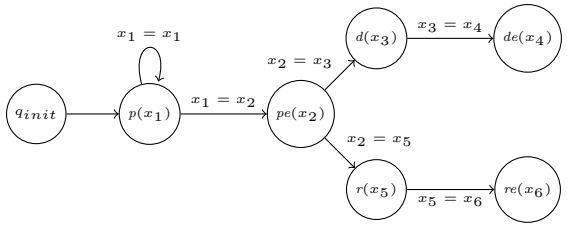


Figure 6: Example of pattern automaton

is non-deterministic both with respect to the state and the valuation of its variables.

An automaton for our running example is represented in Figure 6. The edges represent the pairs for which δ is defined, and the patterns in Υ check the following:

- $\Upsilon(q_{init})$ checks nothing.
- $\Upsilon(\mathtt{p})(x_1)$ checks that the call to `Bill` of the `MailOrder` of `Order-Id` x_1 has been activated and the product is in the catalog. The calls to `Deliver` and to `Reject` are still not activated.
- $\Upsilon(\mathtt{pe})(x_2)$ checks that the call to `Bill` of the `MailOrder` of `Order-Id` x_2 has returned a payment.
- $\Upsilon(\mathtt{d})(x_3)$ checks that the call to `Deliver` of the `MailOrder` of `Order-Id` x_3 is activated and the amount brought by `Bill` is the same as the price of the item that has been ordered.
- $\Upsilon(\mathtt{de})(x_4)$ checks that the call to `Deliver` of the `MailOrder` of `Order-Id` x_4 has been returned.
- $\Upsilon(\mathtt{r})(x_5)$ checks that the call to `Rejection` of the `MailOrder` of `Order-Id` x_5 is activated and the amount brought by `!Bill` is different from the price of the item that has been ordered.
- $\Upsilon(\mathtt{re})(x_6)$ checks that the call to `Rejection` has been returned for the `MailOrder` of `Order-Id` x_6.

We note that in some specification models, such as statecharts [25], states are defined in a hierarchical manner, i.e. entering a state may trigger a more refined state-transition sub-system. Other systems further extend this with *recursion* [9]. Although not done here, one could extend our formalism to capture such hierarchical or recursive states.

Past-Tree-LTL. Finally, we consider workflow constraints specified using temporal formulas. Intuitively, these state, given a particular history, whether a given transition is allowed. The language is a variant of Tree-LTL [5] using only past LTL operators, that we call Past-Tree-LTL. It is obtained from classical propositional LTL (e.g., see [19]) by interpreting each proposition as a parameterized tree pattern $P(\bar{X})$ where $\bar{X}$ is a subset of its variables, designated as *global*. All global variables are treated as free in the patterns and are quantified existentially at the end. The past temporal operators are $\mathbf{X}^{-1}$ (previously) and $\mathbf{S}$ (since), with the standard semantics. Specifically, $\mathbf{X}^{-1}\varphi$ holds for a prerun $(I_0, e_0) \ldots, (I_n, e_n)$ if φ holds at $(I_0, e_0) \ldots, (I_{n-1}, e_{n-1})$; $\varphi \mathbf{S} \psi$ holds at $(I_0, e_0) \ldots, (I_n, e_n)$ if ψ holds in $(I_0, e_0) \ldots, (I_j, e_j)$

for some $j \leq n$ and φ holds in $(I_0, e_0) \ldots, (I_k, e_k)$ for every k, $j < k \leq n$. In summary, a *Past-Tree-LTL formula* is of the form $\exists \overline{X} \psi(\overline{X})$ where ψ uses only the temporal operators $\mathbf{X}^{-1}$ and $\mathbf{S}$, and $\overline{X}$ is the set of global variables of the parameterized patterns interpreting the propositions. The set of Past-Tree-LTL formulas is denoted $\mathcal{T}$. A *TAXML schema* is an expression $S|\theta$ for S a BAXML schema and $\theta \in \mathcal{T}$. A prerun ρ satisfies $\exists \overline{X} \psi(\overline{X})$ if ρ satisfies $\psi(\nu(\overline{X}))$ for *some* valuation ν of the global variables $\overline{X}$ in the active domain of ρ.

The choice to existentially quantify the global free variables appears natural for specifying workflow transition constraints. Observe that such variables are quantified *universally* in the language Tree-LTL of [5], used to specify properties of all runs. However, the model checking approach of [5] is based on checking unsatisfiability of the negation of Tree-LTL formulas, whose global variables then become *existentially* quantified.

To illustrate Past-Tree-LTL constraints, consider the description of valid transitions in the MailOrder example. This can be specified by a Past-Tree-LTL disjunctive formula. One of its disjuncts is the following:

$$\exists y \big(\psi_{?Bill}(y) \wedge \mathbf{X}^{-1} \psi_{!Bill}(y) \wedge \mathbf{X}^{-1} \psi_{\gamma_c(Bill)}(y) \big)$$

stating the existence of an order id y for which $?Bill$ is present in the current instance, $!Bill$ is present in the previous instance, and the guard of $Bill$ is true in the previous instance. This is done using appropriate parameterized patterns[4] $\psi_{?Bill}(y), \psi_{!Bill}(y)$ and $\psi_{\gamma_c(Bill)}(y)$.

Checking workflow constraints

The following establishes the complexity of testing workflow constraints.

THEOREM 4. *Let $S|W$ be a fixed workflow schema, for $W \in \{\mathcal{G}, \mathcal{A}, \mathcal{T}\}$, and ρ a prerun of S. Checking whether ρ satisfies W can be done in* PTIME *with respect to $|\rho|$.*

A more difficult decision problem is checking the *existence* of a valid transition extending the current prerun. Indeed, this is undecidable even for BAXML schemas with no workflow constraints (with either flavor of the abort semantics). The difficulty arises from the power of external functions. Indeed, without external functions it suffices to test all possible call activations and returns. However, the problem becomes decidable for bounded trees.

THEOREM 5. *(i) Given a BAXML schema S and a prerun ρ of S, it is undecidable whether ρ is blocking. (ii) Given a BAXML schema S with non-recursive DTD and a prerun ρ of S, it is decidable whether ρ is blocking.*

5. EXPRESSIVENESS

In this section we compare the expressive power of BAXML, GAXML, AAXML, and TAXML, using the framework developed in Section 2. We begin by comparing the languages relative to views retaining full information about the current BAXML document, that we refer to as identity views. We then consider a more permissive version allowing to hide some of the data and functions, thus providing more leeway for simulations.

[4]The parametrized pattern formula $\psi_{\gamma_c(Bill)}(y)$ is obtained by replacing in $\gamma_c(Bill)$ each label *self* by $!Bill$ and mapping y to the `Order-Id` of the `MailOrder` to which $!Bill$ belongs.

Workflow system semantics

We begin by casting the semantics of BAXML, GAXML, AAXML, and TAXML in terms of the workflow systems described in Section 2. For each specification S (for BAXML) or $S|W$ (for GAXML, AAXML and TAXML), the nodes of the workflow system are the finite prefixes of runs of S or $S|W$. The state label for each node is the last instance in the prefix. The root is the empty prerun, denoted Φ. There is an edge labeled e from node ν to ν' if ν' extends ν with a single instance by event e that is a function call or the return of a such a call. Note that the infinite paths in the tree starting from Φ correspond to the runs of $S|W$. Because of the semantics of blocking runs, each path is extensible to an infinite path.

Note that there are alternative choice of workflow system semantics, and different goals may require different choices. For example, for AAXML it may be natural to retain in the state, information on the current state of the associated automaton together with the valuation of its parameters. This would simplify defining views where such states are included in the observables.

Comparison with identity views

We first compare BAXML, GAXML, AAXML, and TAXML relative to the identity view on the states and events of the workflow system (denoted id), thus preserving full information on the system. Observe that if a language $\mathcal{W}_2$ *simulates* $\mathcal{W}_1$ with respect to (id, id), this means that for each W_1 in $\mathcal{W}_1$, there exists W_2 in $\mathcal{W}_2$, such that $W_1 \sim W_2$, i.e., W_1 and W_2 have exactly the same runs. So, this is a very strong requirement. Note also, that since id is the most refined possible view of a workflow system, existence of simulation with respect to id would imply, by Lemma 1, the existence of simulation with respect to any coarser view. Unfortunately (but not surprisingly), the three extensions of BAXML models are incomparable relative to the identity view.

THEOREM 6. *The workflow languages GAXML, AAXML and TAXML are incomparable relative to $\hookrightarrow_{(id,id)}$.*

Comparison with projection views

Given the negative result of Theorem 6, we next consider simulation relative to views allowing more leeway in the simulating system. Specifically, the view remains the identity on the simulated system, but allows the simulating system to use additional data and functions. We refer to the latter as a projection view and denote the class of projection views by π.

Specifically, let S be a BAXML schema and Σ_0 and $\mathbf{F}_0$ be subsets of the tags and functions of S (the visible symbols) such that, in every instance satisfying the DTD of S, whenever a node has tag $a \notin \Sigma_0$, none of its descendants has a label in Σ_0 or in $\mathbf{F}_0$. The projection $\pi_{\Sigma_0, \mathbf{F}_0}([S])$ is defined as follows. For a state I of $[S]$ (and for any instance), the projection is obtained by removing all nodes whose label is a tag not in Σ_0 or a function not in $\mathbf{F}_0$ and their descendants. We also remove the workspaces whose corresponding function calls have been projected out. The projection of an event $!f(F)$ is ϵ for $f \notin \mathbf{F}_0$ and $!f(\pi_{\Sigma_0, \mathbf{F}_0}(F))$ for $f \in \mathbf{F}_0$, and similarly for $?f(F)$. The projection view is defined in the same way for BAXML augmented with constraints (GAXML, AAXML, and TAXML).

Simulation	Schema blowup	Instance blowup	Silent transitions
GAXML $\hookrightarrow_{(id,\pi)}$ BAXML	exponential	linear in instance	linear in prerun
AAXML^{sib} $\hookrightarrow_{(id,\pi)}$ BAXML	exponential	polynomial in instance	polynomial in prerun
TAXML^{sib} $\hookrightarrow_{(id,\pi)}$ BAXML	exponential	polynomial in prerun	polynomial in prerun
TAXML $\hookrightarrow_{(id,\pi)}$ AAXML	exponential	polynomial in prerun	polynomial in prerun
AAXML $\hookrightarrow_{(id,\pi)}$ TAXML	polynomial	polynomial in instance	O(1)

Figure 7: Cost of various simulations from Theorems 7 and 8

Our main result is that, with projection views, the powerful control mechanisms of GAXML can be simulated by BAXML alone. For AAXML and TAXML, we need a minor restriction forbidding the presence of sibling calls to the same external function (this can be enforced by the DTD). We denote these restrictions by AAXML^{sib} and TAXML^{sib}.

THEOREM 7. $\mathcal{W} \hookrightarrow_{(id,\pi)}$ BAXML
for $\mathcal{W} \in \{\text{GAXML}, \text{TAXML}^{sib}, \text{AAXML}^{sib}\}$.

Since BAXML is included in GAXML, TAXML^{sib}, and AAXML^{sib}, it follows that the four languages can simulate each other relative to projection views.

For AAXML and TAXML, we have the following.

THEOREM 8. *AAXML* $\hookrightarrow_{(id,\pi)}$ *TAXML and* *TAXML* $\hookrightarrow_{(id,\pi)}$ *AAXML.*

The proofs of the above results (omitted) provide insight into the simulations of the four languages by each other, and in particular into the power of imposing control using static constraints. In terms of the cost of each simulation, several parameters can be considered: (i) the blowup in the schema size, (ii) the blowup in the instance size, (iii) the number of silent transitions needed to simulate a single transition. For the simulations considered here, the blowup in the schema size varies from polynomial to exponential, the blowup in the instance size from polynomial with respect to the instance to polynomial with respect to the entire prerun, and the number of silent transitions from constant to polynomial in the prerun (for fixed schemas). The costs for various simulations are spelled out in more detail in Figure 7.

The difficulty of simulating AAXML and TAXML with sibling external function calls by BAXML (or GAXML) lies in the fact that the constraints of AAXML and TAXML must be checked after *every* transition, and GAXML cannot prevent multiple returns from sibling external function calls that skip validity checks. Indeed, as shown below, this difficulty cannot be circumvented.

THEOREM 9. $\mathcal{W} \not\hookrightarrow_{(id,\pi)}$ *GAXML*
for $\mathcal{W} \in \{\text{TAXML, AAXML}\}$.

Comparison with coarser views

Theorem 7 shows that BAXML, GAXML, TAXML^{sib} and AAXML^{sib} can simulate each other relative to projection views. This result turns out to be quite powerful. Indeed, by Lemma 1 it implies that the simulation results can be extended to any views that are coarser than such views. For example, one may wish to focus on the sequence of events (function calls and returns, together with their arguments), ignoring state information. This information can be captured by composing the views in id and π with a view V that is the identity on events and maps every state to a fixed constant. By Lemma 1, BAXML, GAXML, TAXML^{sib} and AAXML^{sib} can simulate each other relative to $(id \circ V, \pi \circ V)$. Similar remarks apply to TAXML and AAXML.

Conversely, one may be interested in observing certain characteristics of the *states* in the tree of runs, ignoring event information. Once again, this can be captured by coarser views than (id, π) so the four languages can simulate each other relative to such views.

6. BAXML AND TUPLE ARTIFACTS

In the previous section, we compared the expressiveness of several workflow languages centered around the common core provided by BAXML. In this section, we illustrate how views can be used to reconcile models that are otherwise incomparable. For this, we use the views framework to compare BAXML workflows with *tuple artifacts* workflows, a variant of IBM's Business Artifacts, which uses relational databases as its underlying model. The main result is that BAXML can simulate tuple artifacts. Indeed, tuple artifacts can be seen as views of BAXML. We will also see that tuple artifacts cannot simulate BAXML even with respect to coarse views retaining just the traces of service and function calls.

We first review informally the tuple artifact model, as presented in [16] (see Appendix for the formal definition). We denote the model by $\mathcal{TA}$. We assume an infinite data domain D. An artifact system consists of a set of artifacts and a set of services acting on the artifacts. An artifact consists of an *artifact tuple* and a set of *state relations*. In addition, an artifact system has an underlying database shared by all artifacts and services, that is fixed throughout a run of the system.

Each service causes a modification of one or several current artifacts. Intuitively, the focus is on the evolution of the artifact tuples, while the state relations are used to carry auxiliary information needed by the services. A service consists of the following:

- a pre-condition, which is an FO formula on the set of artifacts of the system and the underlying database;
- a post-condition, which is an FO formula on the set of artifacts and the database, defining, for each artifact tuple, the values allowed in the next instance; free variables range over the infinite domain D, so may take new values not present in the current instance;
- for each state relation, two FO formulas defining the sets of tuples to be inserted and deleted from the state. The formulas take as input the current artifact instance and the database, and are interpreted with active domain semantics. Thus, their result is always finite.

Services are applied non-deterministically. At any given time, a service can be applied to the current instance if its pre-condition holds and if the post-condition is satisfiable. Thus, there are two forms of non-determinism in a transition: one stemming from the choice of service, and another from the choice of values for the next artifact tuples, from among those satisfying the post-condition. A *run* of an artifact system is a sequence of consecutive instances together with the name of the service applied at each transition. (For initial instance, we take any instance whose artifact states are empty.) As for BAXML, blocking runs are extended by repeating forever the last configuration, with the corresponding transitions labeled by the special event *block*. See [16] for a detailed example of an artifact system.

In order to simulate $\mathcal{TA}$ with BAXML, we must define views that render the two compatible. For $\mathcal{TA}$, we simply take the identity views *id*. For BAXML, we consider schemas of a special form, that represent the artifact tuples and relations (states and database) of $\mathcal{TA}$. A relation R with attributes $A_1 \dots A_m$ is naturally represented in BAXML by a subtree rooted at R, satisfying the DTD:

$$\begin{array}{lcl} R & \rightarrow & |tup_R| \geq 0 \\ tup_R & \rightarrow & \wedge_{i=1}^{m} |A_i| = 1 \\ A_i & \rightarrow & |dom| = 1 \end{array}$$

We will have to record several instances of a state relation R. To distinguish the current one, it will be adorned by a function call !*current* just placed under its root, i.e., an R-labeled node. Artifact tuples are handled similarly. Each service of the artifact system is modeled in BAXML by a corresponding function with the same name. The call of a service is captured in BAXML by a call to the corresponding function. Given a BAXML instance as described, the view is defined as follows. On states of the BAXML workflow system, the view maps the subtrees representing database and state relations, and artifact tuples, to the corresponding relations and tuples. Events consisting of activations of functions corresponding to services are mapped to the corresponding service name, and all others are mapped to ϵ. We denote this class of views by $\mathcal{V}_{\mathcal{TA}}$. The main result is the following.

THEOREM 10. $\mathcal{TA} \hookrightarrow_{(\mathrm{id}, \mathcal{V}_{\mathcal{TA}})}$ *BAXML.*

Thus, BAXML can simulate $\mathcal{TA}$. In fact, since the view used for $\mathcal{TA}$ is the identity, tuple artifacts themselves can be seen as views of BAXML systems. The simulation yields a BAXML schema polynomial in the $\mathcal{TA}$ schema, BAXML instances polynomial in the $\mathcal{TA}$ instances, and polynomially many silent transitions (with respect to the current instance), to simulate in BAXML one transition of $\mathcal{TA}$.

Conversely, we will show that, in a strong sense, $\mathcal{TA}$ cannot effectively simulate BAXML. We use coarse views that retain just the names of function calls in BAXML and of service calls in $\mathcal{TA}$ (modulo a projection). Such views are natural because the traces of function and service calls largely capture the sequencing of events central to workflows. We will prove a strong negative result for such views. Intuitively, the problem in simulating BAXML with $\mathcal{TA}$ is due to the fact that BAXML can read a large structure (for example an entire relation represented as an XML document) by a single function call. On the other hand, tuple artifacts can only read one tuple at a time, so the simulation requires a loop. This loop may lead to an infinite sequence of ϵ-transitions (imagine a denial-of-service attack in which the attacker keeps sending new tuples). But if no such sequence of ϵ-transitions occurs in the BAXML system, this is not a correct simulation.

More precisely, the views we use are defined as follows:

states for both BAXML and $\mathcal{TA}$, all states are mapped to a constant state (so all information about the states is lost);

events for BAXML, active calls $?g$ are mapped to ϵ and calls $!g$ are mapped to g or to ϵ (so some function calls can be hidden); for $\mathcal{TA}$, a service σ is mapped to σ or to ϵ (so again, some services can be hidden).

We denote the above class of views of BAXML systems by $\mathcal{V}_{fun}$ and of $\mathcal{TA}$ systems by $\mathcal{V}_{serv}$.

Recall that the definition of simulation does not require effective construction of the simulating schema (even though all our positive simulation results are constructive). We can show that one cannot effectively construct a $\mathcal{TA}$ specification simulating a given BAXML schema, with respect to the above views.

THEOREM 11. *There is no algorithm that, given as input a BAXML schema W_1 and a view $V_1 \in \mathcal{V}_{\mathrm{fun}}$ produces a $\mathcal{TA}$ schema W_2 with a view $V_2 \in \mathcal{V}_{\mathrm{serv}}$ such that $V_1([W_1]) \sim V_2([W_2])$. Moreover, this holds even for BAXML schemas of bounded depth.*

The proof (using only BAXML schemas of bounded depth) relies on the undecidability of implication for FDs and IDs.

REMARK 2. *By Lemma 1 (applied to effective simulations), the negative result of Theorem 11 extends to any views that expose* more *information than those above.*

7. CONCLUSION

This paper makes a dual contribution. First, it proposes a flexible framework for comparing distinct workflow models by means of views extracting a common set of observable states and events, and a natural notion of simulation. Second, it uses this framework to compare concrete languages capturing some of the main workflow specification paradigms: automata, temporal constraints, and pre-and-post conditions. These were first investigated using as a common core BAXML, where the integration of XML and embedded function calls allows to naturally support a wide range of data-centered tasks. We proved the surprising result that the static constraints of BAXML are alone sufficient to simulate the three apparently much richer workflow specification languages mentioned earlier. Beyond the specifics of the XML-based model, the results provide insight into the power of the various workflow specification paradigms, the trade-offs involved in choosing one over another, and the relation to static constraints. Finally, we compared BAXML to tuple artifacts, a variant of IBM's Business Artifact model using relational databases. We showed that BAXML can simulate tuple artifacts whereas the converse is false. To compare these very different models, we used again the views framework to render them compatible. This illustrates the usefulness of the view-based framework to reconcile seemingly incomparable workflow models.

8. REFERENCES

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

[2] S. Abiteboul, P. Bourhis, A. Galland, and B. Marinoiu. The axml artifact model. In *Time*, 2009.

[3] S. Abiteboul, L. Herr, and J. V. den Bussche. Temporal versus first-order logic to query temporal databases. In *PODS*, 1996.

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

[5] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *TODS*, 2009. Extended abstract in PODS 08.

[6] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000. Extended abstract in PODS 98.

[7] N. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.

[8] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. *JCSS*, 66(4):688–727, 2003.

[9] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[10] M. Arenas, W. Fan, and L. Libkin. Consistency of XML specifications. In *PODS*, 2002.

[11] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, 2007.

[12] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, 2006.

[13] BPEL. `http://bpel.xml.org/`.

[14] D. Calvanese, G. D. Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In *ICSOC/ServiceWave*, pages 130–143, 2009.

[15] S. Demri and R. Lazić. Ltl with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3):1–30, 2009.

[16] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, 2009.

[17] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *JCSS*, 73(3):442–474, 2007.

[18] G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou. A framework for optimizing distributed workflow executions. In *DBLP*, 1999.

[19] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. MIT Press, 1990.

[20] W. Fan and L. Libkin. On XML integrity constraints in the presence of dtds. In *PODS*, 2001.

[21] C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *ICDT*, 2009.

[22] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow infrastructure management. *Distributed and Parallel Databases*, 3:119–153, 1995.

[23] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *IEEE International Conference on Service-Oriented Computing and Applications*, 2007.

[24] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, 2007.

[25] D. Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program*, 8(3):231–274, 1987.

[26] R. Hull. Personal communication, 2009.

[27] R. Hull, F. Llirbat, B. Kumar, G. Zhou, G. Dong, and J. Su. Optimization techniques for data-intensive decision flows. In *ICDE*, 2000.

[28] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration*, 1999.

[29] D. Martin et al. OWL-S: Semantic markup for web services, W3C Member Submission, November 2003.

[30] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[31] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[32] W. Mok and D. Paper. Using Harel's statecharts to model business workflows. *J. of Database Management*, 13(3):17–34, 2002.

[33] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, 2002.

[34] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.

[35] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[36] L. Segoufin. Static analysis of XML processing with data values. *SIGMOD Record*, 36(1):31–38, 2007.

[37] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS*, 66(1):40–65, 2003.

[38] J. van Benthem. *Modal correspondence theory*. PhD thesis, Mathematish Instituut & Instituut voor Grondslagenonderzoek, Univ. of Amsterdam, 1976.

[39] W. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management, 2004. In *Lectures on Concurrency and Petri Nets*.

[40] W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[41] W. M. P. van der Aalst and A. H. M. ter Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In *Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, 2002*, 2002.

[42] J. Wang and A. Kumar. A framework for document-driven workflow systems. In *Business Process Management*, pages 285–301, 2005.