Comparing Workflow Specification Languages: A Matter of Views

Serge Abiteboul, INRIA Saclay, ENS Cachan and Univ. Paris-Sud
Pierre Bourhis, Univ. Paris-Sud and Univ. of Oxford
Victor Vianu, Univ. of California, San Diego

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 [BPEL]. 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 ([Hull 2009]).

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 lan-

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. P. Bourhis was supported in part by EP/G004021/1 and EP/H017690/1 of the Engineering and Physical Sciences Research Council UK. V. Vianu was supported in part by the National Science Foundation under grant IIS-0916515.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

guages. 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 pertaining 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 workflow 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 [Abiteboul et al. 2008], 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 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 [Abiteboul et al. 2008], 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 [Deutsch et al. 2009]. 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., [Georgakopoulos et al. 1995; van der Aalst 2004]). This has been captured in the workflows community by flowcharts, Petri nets [van der Aalst 1998; van der Aalst and ter Hofstede 2002; Adam et al. 1998], and state charts [Harel 1987; Mok and Paper 2002]. The comparison of such systems using the notion of bisimulation is considered in [Milner 1989; van Benthem 1976]. More recently, data-centric workflows have been considered in [Wang and Kumar 2005], and in particular the *artifact* model of IBM [Nigam and Caswell 2003]. Verification for such models is considered in [Gerede et al. 2007; Gerede and Su 2007; Bhattacharya et al. 2007; Deutsch et al. 2009; Fritz et al. 2009]. The comparison of such systems is considered in [Calvanese et al. 2009] 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 [Hull et al. 1999; Dong et al. 1999; Hull et al. 2000], the OWL-S proposal [McIlraith et al. 2001; Martin et al. 2003] as well as some work on semantic Web services [Narayanan and McIlraith 2002]. The article [Deutsch et al. 2007] (building on [Spielmann 2003; Abiteboul et al. 2000]), 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 [Emerson 1990; Abiteboul et al. 1996; Spielmann 2003]. Apart from the work on verification of BAXML with guards mentioned above [Abiteboul et al. 2008], most other work on static analysis on XML (with data values) deals with documents that do not evolve in time, e.g., [Fan and Libkin 2001; Arenas et al. 2002; Alon et al. 2003]. This motivated studies of automata and logics on strings and trees over infinite alphabets [Neven et al. 2004; Demri and Lazić 2009; Bojanczyk et al. 2006]. See [Segoufin 2007] for a survey on related issues.

A survey on Active XML may be found in [Abiteboul et al. 2008]. In [Abiteboul et al. 2009], 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.

This paper is the extended version of the conference article [Abiteboul et al. 2011]. It differs from the latter by including the full technical development, including the proofs.

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.

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 infinite 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 2.1. [Workflow System] A workflow system is a tuple $\mathit{WS} = (N, n_0, \delta, q_0, \lambda_N, \lambda_\delta)$ 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 $\lambda_\delta((n, n')) = \mathit{block}$ 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, defined in Section 4, 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 workflow 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 $\mathit{WS} = (N, n_0, \delta, q_0, \lambda_N, \lambda_\delta)$ and V be a view. Then $V(\mathit{WS})$ 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 $V(\mathit{WS})$ 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 systems, 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 [Milner 1989] (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.

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 $\xrightarrow{e}$ on nodes by $n \xrightarrow{e} m$ 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 on 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 2.2. [w-bisimulation] Let $\mathit{WS}_i = (N^i, n_0^i, \delta^i, q_0, \lambda_N^i, \lambda_\delta^i)$, $i \in \{1, 2\}$, 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 there exists n_1' such that $n_1 \xrightarrow{e} n_1'$ in WS_1 then there exists n_2' such that $n_2 \xrightarrow{e} n_2'$ in WS_2 and $B(n_1', n_2')$,
— For each event $e \neq \epsilon$, if there exists n_2' such that $n_2 \xrightarrow{e} n_2'$ in WS_2 then there exists n_1' such that $n_1 \xrightarrow{e} n_1'$ in WS_1 and $B(n_1', n_2')$.
— 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 $\mathit{WS}_1 \sim \mathit{WS}_2$ the fact that there exists a w-bisimulation of WS_1 and WS_2.

The last condition captures the intuition that progress from a given state along a path in the simulated system must imply progress from the corresponding state along a path in the simulating system, where progress means the occurrence of a non-silent event. We note that there are well-known notions of bisimulation related to ours, such as

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

weak-bisimulation and observation-congruence equivalence, motivated by distributed algebra [Milner 1989]. 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 2.3. [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 by $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1,\mathcal{V}_2)} \mathcal{W}_2$, 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 2.4. 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 6.9 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 2.5. [*Composition Lemma*] *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 2.6. *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 [Abiteboul et al. 2008], 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.

We begin with an informal overview of the model, then provide more details. To illustrate our definitions, we use a simplified version of the Mail Order example of [Abiteboul et al. 2008]. 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.

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. More precisely, a function symbol !`f` indicates a node where function `f` can be called, and a function symbol ?`f` indicates that a call to `f` has been made but the answer has not yet been returned. For example, a BAXML document is shown in Figure 1. The BAXML document may be subject to static constraints specified by a DTD, as well as Boolean combinations of tree patterns. For example, the negation of the pattern in Figure 2 (a) says that an Order ID uniquely determines the product and customer names. In patterns, double edges denote descendant and single edges the child relation.

A BAXML document evolves as a result of making function calls and receiving their results. A call can be made and an answer can be returned at any point, as long as the resulting instance satisfies the static constraints. The argument of the call is specified by a query on the document, producing a forest. The query may refer to the node at which the call is made (denoted by *self*), so the location of the call in the document is important. When a call is made at node x labeled !f, its label changes from !f to ?f. The result of a call consists of another BAXML document, so a forest, whose trees are added as siblings of the node x where the call was made. After the answer of the call at node x is returned, x may be kept (in which case its label reverts to !f) or x may be deleted. This is specified by the schema, for each function f. If calls to !f are kept, f is called *continuous*, otherwise it is *non-continuous* (this is specified in the schema).

For example, consider the MailOrder function in Figure 1. Intuitively, its role is to fetch new mail orders from customers. For instance, one result of a call to !MailOrder may consist of the subtree with root MailOrder in Figure 1. Since the function is processed externally, the semantics of its evaluation is not known. We call such a function *external*. Its specification consists of its input query and a DTD constraining the allowed answers. In addition to external functions, there are functions processed internally by the BAXML system. These are called *internal*. For example, Bill is such a function. When a call to Bill is made at a node x labeled !Bill, the label of x turns to ?Bill (to indicate that a call has been made whose answer is still pending) and the call is processed internally. Specifically, the call generates a new BAXML document (a *workspace*) that evolves under function calls and returns. The answer can be returned at any point when the workspace contains no running calls (i.e. no nodes labeled ?g for some g) and the resulting instance satisfies the static constraints. The contents of the result is specified by a *return query* that applies to the workspace. For example, the answer to a call to Bill can be returned once payment has been received. The answer, specified by the return query, provides the product paid for and amount of payment (see Example 3.1).

Once the result of a call has been returned, the BAXML document of the corresponding workspace is removed. In order for the result to be returned at the correct location (next to node x), a mapping called *eval* is maintained between nodes where calls have been made and BAXML document corresponding to the workspaces (e.g., see Figure 5). The system evolves by repeated function calls and answer returns, occurring one at a time non-deterministically. This may reach a *blocking instance* in which no function can be called and no result can be returned, or may continue forever, leading to an infinite run. For example, runs of the Mail Order system are always infinite since new mail orders can always be fetched. For uniformity, we make all runs infinite by repeating blocking instances forever.

We now describe the BAXML model in more detail. We assume given the following disjoint infinite sets: *nodes* $\mathcal{N}$ (denoted by n, m), *tags* Σ (denoted by $a, b, c, \ldots$), *function names* $\mathcal{F}$, *data values* $\mathcal{D}$ (denoted by $\alpha, \beta, \ldots$) *data variables* $\mathcal{V}$ (denoted by $X, Y, Z, \ldots$), possibly with subscripts. In the model, trees are unranked and unordered.

For each function name f, we also use the symbols !f and ?f, called *function symbols*, and denote by $\mathcal{F}^!$ the set $\{!\mathtt{f} \mid \mathtt{f} \in \mathcal{F}\}$ and by $\mathcal{F}^?$ the set $\{?\mathtt{f} \mid \mathtt{f} \in \mathcal{F}\}$. As described above, !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). When a call to f is made at a node x labeled !f, the label changes from !f to ?f. After the answer of the call at node x is returned, the node x may be kept or the node x may be deleted. If x is kept, its label changes from ?f back to !f. 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.

Fig. 1: A BAXML document.

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 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, it is natural to require that `Order-Id` 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 a 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 quantified existentially in P and P', independently of each other.

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 by $I, \nu \models P(\bar{X})$, if I satisfies the pattern $P(\nu(\bar{X}))$ obtained by replacing each variable in $\bar{X}$ by its value under ν.

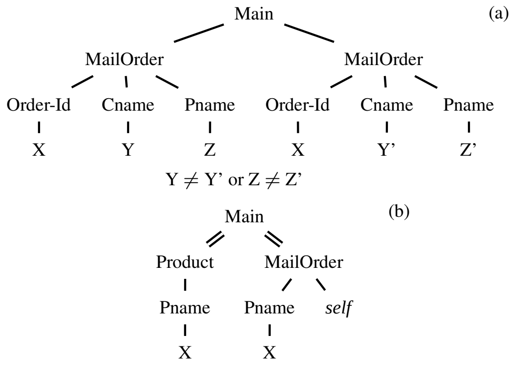


Fig. 2: Two patterns

For convenience, we sometimes use a self-explanatory XPath-like notation to specify simple 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* → *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 a relative query (with a single rule) is given in Figure 3 (the notation *self*:!`Bill` means that the node *self* must be labeled !`Bill`). The label of the constructor node (indicated by brackets) is `Process-bill`.

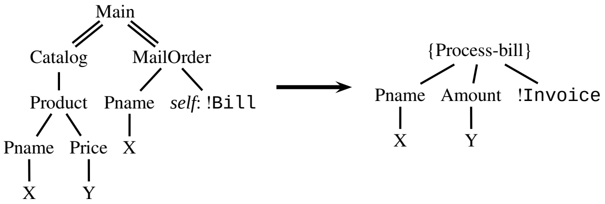


Fig. 3: Example of a relative query

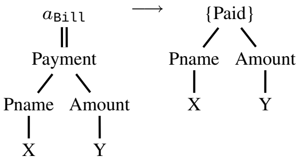
Consider the evaluation of the query of Figure 3 on the BAXML document of Figure 1 at the unique node labeled !`Bill`. There is a unique matching of the *Body* pattern and the result is the *Head* pattern of the query with X replaced by *Nikon* and Y by 199 (without brackets for `Process-bill`).

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

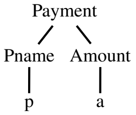
Schemas and instances. A BAXML *schema* S is a tuple $(\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ where (i) the set Φ_{int} contains a finite set of internal function specifications, (ii) the set Φ_{ext} contains a finite set of external function specifications, and (iii) Δ consists of a DTD and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

We next detail Φ_{int} and Φ_{ext}. For each $\mathtt{f} \in \mathcal{F}$, let $a_{\mathtt{f}}$ be a new distinct label in Σ. Intuitively, $a_{\mathtt{f}}$ labels the roots of all workspaces resulting from calls to `f`. The specification of a function `f` of Φ_{int} indicates whether `f` is continuous or not, provides its *argument query* , and *return query* . The role of the argument query is to define the initial state of the workspace generated by the call to `f`. The argument query is a relative query. When the query is evaluated, *self* binds to the node at which the call !`f` is made. The return query applies to the current state of the workspace corresponding to the call. Thus, it is a query in which every tree pattern occurring in the body of a rule is rooted at $a_{\mathtt{f}}$. .

Example 3.1. 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. The return query of `Bill` is:



Intuitively, this makes sense assuming that the function `Invoice` returns a tree of the form:



where p is the product name and a the amount of payment.

Each function `f` in Φ_{ext} is specified similarly, except that the return query is missing. In addition, a DTD $\Delta_{\mathtt{f}}$ 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 $\Delta_{\mathtt{f}}$ 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 $\mathtt{f} \in \Phi_{\text{int}}$ such that: (i) for each n with label

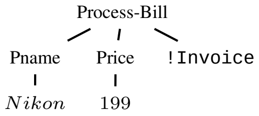


Fig. 4: Answer of the query in Figure 3 applied to the instance in Figure 1

$?\mathtt{f}$, *eval*(n) is a tree in $\mathcal{T}$ with root label $a_{\mathtt{f}}$ (its workspace), and (ii) every tree in $\mathcal{T}$ with root label $a_{\mathtt{f}}$ is *eval*(n) for some n labeled $?\mathtt{f}$. An instance of S is *valid* if it satisfies Δ. More precisely, each tree in the forest making up the instance satisfies the DTD of Δ, and the instance as a whole satisfies the Boolean combination of patterns of Δ.

Runs. Let $I = (\mathcal{T}, \mathit{eval})$ and $I' = (\mathcal{T}', \mathit{eval}')$ 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 $!\mathtt{f}(F)$ or $?\mathtt{f}(G)$, where:

— $\mathtt{f}$ is a function;
— F is the forest consisting of the result of applying the argument query of $\mathtt{f}$ to $\mathcal{T}$, at some node labeled $!\mathtt{f}$;
— G is the forest consisting of the answer to a running call $?\mathtt{f}$ at some node n. More precisely, if $\mathtt{f}$ is internal, G is the result of applying the return query of $\mathtt{f}$ to *eval*(n). If $\mathtt{f}$ is external, G is any forest satisfying the DTD Δ_f for answers of $\mathtt{f}$.

For technical reasons, we also use two special events, *init* that only generates the initial instance, and *block*, whose use will be clear shortly. Initial instances of BAXML schemas are defined below. 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. Consider $I = (\mathcal{T}, \mathit{eval})$ and an event $!\mathtt{f}(F)$, resulting from a call to $!\mathtt{f}$ at some node n of $\mathcal{T}$. The next instance, if it exists, is the instance $I' = (\mathcal{T}', \mathit{eval}')$ satisfying Δ, obtained as follows:

— change the label of n to $?\mathtt{f}$
— if $\mathtt{f}$ is internal, add to the graph of *eval* the pair (n, T') where T' is a tree consisting of a root $a_{\mathtt{f}}$ connected to all trees in F (the result of evaluating the argument query of f on input $(\mathcal{T}, n)$).

If the resulting instance does not satisfy Δ, there is no next instance under the event $!\mathtt{f}(F)$.

Now consider an event $?\mathtt{f}(G)$, resulting from returning the answer G of a running call $?\mathtt{f}$ at some node n of $\mathcal{T}$. Recall that, if $\mathtt{f}$ is internal and *eval*(n) contains no running function calls, G is the result of applying the return query of $\mathtt{f}$ to *eval*(n). If $\mathtt{f}$ is external, G is any forest satisfying $\Delta_{\mathtt{f}}$. Then the instance I' is obtained as follows:

— add all trees in G as siblings to n
— if $\mathtt{f}$ is internal, remove $(n, \mathit{eval}(n))$ from the graph of *eval* and the tree *eval*(n) from $\mathcal{T}$
— if $\mathtt{f}$ is non-continuous, remove the node n from $\mathcal{T}$
— if $\mathtt{f}$ is continuous, change the label of n from $?\mathtt{f}$ to $!\mathtt{f}$.

If the resulting instance does not satisfy Δ, then there is no next instance under the event $?f(\mathtt{G})$.

Figure 5 shows a possible next instance for the instance of Figure 1 after an internal call has been made to !Bill. The node associated with this internal call is denoted by n. Recall the specification of Bill from Example 3.1. The argument query of Bill is the query in Figure 3. For each homomorphism from the body (left pattern) of Figure 3 to the document such that the node labeled $self$ is associated with n, a valuation of the variables is defined. In this example, there is one homomorphism defining the following valuation : $X = Nikon$ and $Y = 199$. The answer of the query is built by applying the previous valuation to the variable in the head of the query (the right part). The answer is described in Figure 4. The workspace of the function Bill is built by placing the answer from Figure 4 under a new root n' labeled $a_{\mathtt{Bill}}$. This workspace is added to the current instance and the function *eval* is updated by setting *eval*$(n) = n'$. The resulting instance is shown in Figure 5, where the dotted arrow represents 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 [Abiteboul et al. 2008]. 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 $i > 0$, $I_{i-1} \vdash_{e_i} I_i$. Intuitively, e_0 generates the initial instance I_0. A *run* is an infinite sequence $\rho = \{(I_i, e_i)\}_{i \geq 0}$ such that each finite prefix of ρ is a prerun of

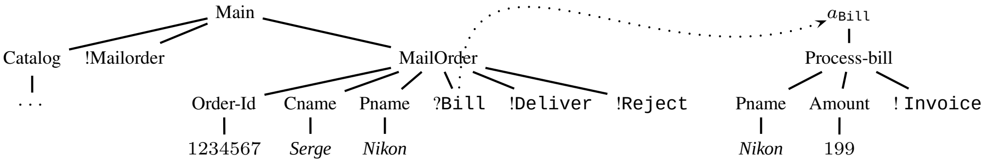


Fig. 5: An instance with an $eval$ link

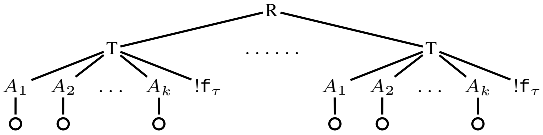


Fig. 6: Relation adorned with some functions

S, or 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*. In the first case the run is called *nonblocking*; in the second case it is called *blocking*.

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 (the non-existence of a legal transition from the blocking instance is ensured by the maximality condition in the definition).

Semantics with and without aborts. We next discuss a subtle difference between the semantics adopted here and that of [Abiteboul et al. 2008]. 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 [Abiteboul et al. 2008] 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 3.2. *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.*

PROOF. The proof for arbitrary DTDs is trivial by the undecidability of satisfiability of static constraints [David 2008]. 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 [Abiteboul et al. 1995]).

Let R be a relation with k attributes, Γ a set of FDs and IDs over R, and F an FD over R. We construct a BAXML schema S and an initial instance I_0 such that $\Gamma \not\models F$ iff there is a valid run from I_0. We represent relation R with attributes $A_1 \cdots A_k$ in the standard way, as a tree rooted at R. Relation R, together with some additional functions whose role will become apparent, is depicted in Figure 6. Clearly, this structure can be enforced by the DTD.

Static constraints can easily require satisfaction of the FDs in Γ and violation of F. In order to check that the inclusion dependencies of Γ are satisfied, we use one internal, non-continuous function $\mathtt{f}_\tau$ for each $\tau \in \Gamma$. One occurrence of each $\mathtt{f}_\tau$ is attached to each tuple of R, as in Figure 6. The functions $\mathtt{f}_\tau$ always return the empty answer. Static constraints require the following:

(i) there is at most one occurrence of $?\mathtt{f}_\tau$ for each τ,
(ii) whenever $?\mathtt{f}_\tau$ occurs, the ID τ is satisfied for the tuple to which $?\mathtt{f}_\tau$ is attached.

The constraint (i) is expressed by conjunctions of negations of patterns as in (i) of Figure 7, and (ii) is enforced by the conjunction of patterns as in (ii) of the same figure, illustrating the case when $\tau = R[A_i] \subseteq R[A_j]$.

Finally, the global DTD specifies a root r, under which one can find either a subtree rooted at R of the shape above, or one external, non-continuous function $!\mathtt{h}$. Thus, the instance I_0 consisting of the root r with child $!\mathtt{h}$ is a possible

[2] There is no (I', e') where $(I_0, e_0), ..., (I_n, e_n)(I', e')$ is a prerun of S.
[3] A DTD is *recursive* if 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.

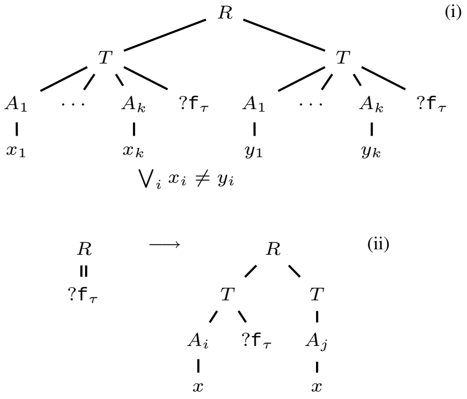


Fig. 7: (i) Pattern whose negation forbids two activated calls and (ii) ensuring satisfaction of $[A_i] \subseteq [A_j]$

initial instance. Note that every valid run of S must end in a blocking instance, in which no function calls occur. Clearly, there exists such a valid run from I_0 iff the function h can return a tree R witnessing that $\Gamma \not\models F$. $\square$

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 (yielding the AAXML model), and (iii) temporal constraints (yielding the TAXML model). 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: each finite prefix of ρ is a prerun of S *that satisfies W*, or 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*. In the first case the run is called *nonblocking*; in the second case it is called *blocking*.

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 forbidden 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 8, 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 $S = (\Phi_{int}, \Phi_{ext}, \Delta)$ 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 at node n of instance $I = (\mathcal{T}, \mathit{eval})$ if $\gamma_c(\mathtt{f})$ holds on $(\mathcal{T}, n)$.

—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 $a_{\mathtt{f}}$ for each internal function f. The result of a call to f at node n of instance $I = (\mathcal{T}, \mathit{eval})$ is returned only when $\gamma_r(\mathtt{f})$ is satisfied on *eval*(n). 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 $\gamma_c(\mathtt{f})$ holds in (I_{i-1}, u), and if the transition results from the return of an internal function call ?f at node u, $\gamma_r(\mathtt{f})$ holds in $eval_{i-1}(u)$. Observe that these constraints involve consecutive instances only.

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

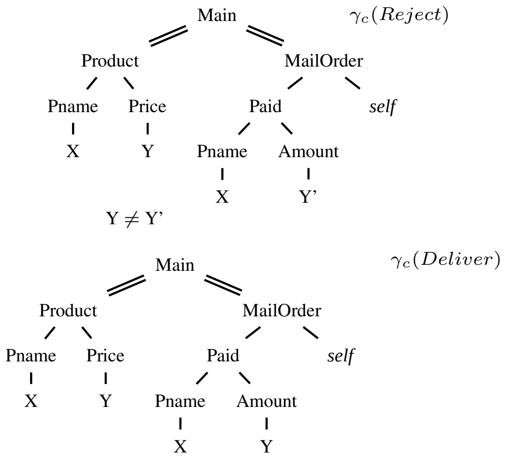


Fig. 8: Call guards of `Reject` and `Deliver`.

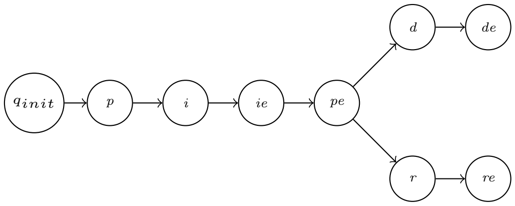
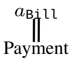


Fig. 9: Example of pattern automaton

Example 4.1. Figure 8 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 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 $q \in Q$, $\Upsilon(q)$ is a Boolean combination of parameterized patterns whose set of free variables equals $\overline{X}_q$;
- the transition function δ is a partial function from $Q \times Q$ such that $\delta(q, q')$ is a Boolean combination of equalities of variables among $\overline{X}_q$ 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.

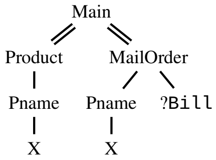
Let $\mathcal{A}$ be the set of pattern automata. An *AAXML schema* is an expression $S|A$ for a BAXML schema S and $A \in \mathcal{A}$. A prerun $\rho = \{(I_i, e_i)\}_{i \leq n}$ of S satisfies an automaton constraint A, denoted by $\rho \models A$, if there exists a sequence $\{(q_i, \nu_i)\}_{i \leq n}$, 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) $\nu_i(\overline{X}_{q_i}) \cup \nu_{i+1}(\overline{X}_{q_{i+1}}) \models \delta(q_i, q_{i+1})$.

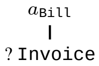
Intuitively, the state of such an automaton after reading a finite sequence ρ of instances is a pair (q, ν) where ν is a valuation of the variables in $\overline{X}_q$. Note that the automaton is non-deterministic both with respect to the state and the valuation of its variables.

Example 4.2. An automaton for our running example is represented in Figure 9. The edges represent the pairs for which δ is defined, and the patterns in Υ check the following:

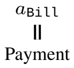
— $\Upsilon(q_{init})$ checks nothing.
— $\Upsilon(\texttt{p})$ checks that the call to `Bill` has been activated and the product is in the catalog:



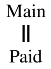
— $\Upsilon(\texttt{i})$ checks that the call to `Invoice` in the workspace of `Bill` has been activated:



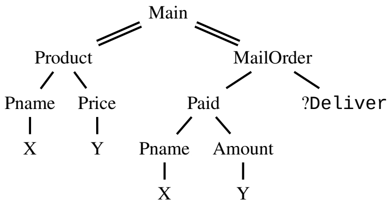
— $\Upsilon(\texttt{ie})$ checks that the call to `Invoice` in the workspace of `Bill` has returned a payment:



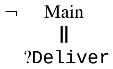
— $\Upsilon(\texttt{pe})$ checks that the call to `Bill` has returned a payment:



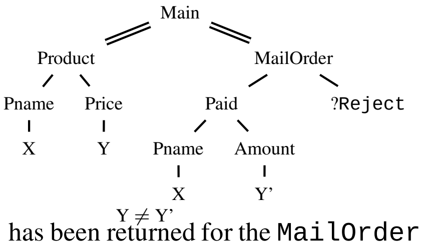
— $\Upsilon(\texttt{d})$ checks that the call to `Deliver` is activated and the amount brought by `Bill` is the same as the price of the item that has been ordered.



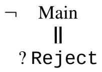
— $\Upsilon(\texttt{de})$ checks that the call to `Deliver` has been returned.



— $\Upsilon(\texttt{r})$ checks that the call to `Reject` is activated and the amount brought by `!Bill` is different from the price of the item that has been ordered:



— $\Upsilon(\texttt{re})$ checks that the call to `Reject` has been returned for the `MailOrder` (there is no active call to `Reject`): .



We note that in some specification models, such as state-charts [Harel 1987], 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* [Alur et al. 2005]. 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 [Abiteboul et al. 2008] using only past LTL operators, that we call Past-Tree-LTL. It is obtained from classical propositional LTL (e.g., see [Emerson 1990]) 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), **S** (since) and $\mathbf{G}^{-1}$ (always previously). The semantics of the different operators is inductively defined as follows:

— $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi$, where φ is a pattern iff I_n satisfies φ.
— $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_1 \wedge \varphi_2$ iff $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_1$ and $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_2$
— $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_1 \vee \varphi_2$ iff $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_1$ or $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi_2$
— $(I_0, e_0), \ldots, (I_n, e_n) \models \neg\varphi$ iff $(I_0, e_0), \ldots, (I_n, e_n) \not\models \varphi$
— $(I_0, e_0), \ldots, (I_n, e_n) \models \mathbf{X}^{-1}\varphi$ iff $(I_0, e_0), \ldots, (I_{n-1}, e_{n-1}) \models \varphi$
— $(I_0, e_0), \ldots, (I_n, e_n) \models \varphi \mathbf{S} \psi$ iff $(I_0, e_0), \ldots, (I_j, e_j) \models \psi$ for some $j \leq n$ and φ holds in $(I_0, e_0), \ldots, (I_k, e_k)$ for every $k, j < k \leq n$
— $(I_0, e_0), \ldots, (I_n, e_n) \models \mathbf{G}^{-1}\varphi$ iff $(I_0, e_0), \ldots, (I_j, e_j) \models \varphi$ for each j, $0 \leq j \leq n$.

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 **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 by $\mathcal{T}$. 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.

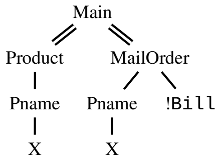
The choice to existentially quantify the global free variables appears natural for specifying workflow transition constraints. Observe that such variables are quantified *universally* in the language Tree-LTL of [Abiteboul et al. 2008], used to specify properties of all runs. However, the model checking approach of [Abiteboul et al. 2008] is based on checking unsatisfiability of the negation of Tree-LTL formulas, whose global variables then become *existentially* quantified.

Example 4.3. 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 conjunctive formula:

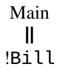
$$\begin{array}{ll}
 & \mathbf{G}^{-1}(\psi_{?\mathit{Bill}} \wedge \mathbf{X}^{-1}(\psi_{!\mathit{Bill}}) \implies \mathbf{X}^{-1}(\psi_{\gamma_c(Bill)})) \\
\wedge & \mathbf{G}^{-1}(\psi_{?\text{Invoice}} \implies \mathbf{X}^{-1}\psi_{?\mathit{Bill}}) \\
\wedge & \mathbf{G}^{-1}(\psi_{\text{Payment}} \implies \mathbf{X}^{-1}\psi_{?\mathit{Invoice}}) \\
\wedge & \mathbf{G}^{-1}(\psi_{\text{Paid}} \wedge \psi_{!\text{Deliver},!\text{Reject}} \implies \mathbf{X}^{-1}\psi_{?\mathit{Payment}}) \\
\wedge & \mathbf{G}^{-1}(\psi_{?\text{Deliver}} \implies \mathbf{X}^{-1}\psi_{\gamma_c(Deliver)}) \\
\wedge & \mathbf{G}^{-1}(\psi_{?\text{Reject}} \implies \mathbf{X}^{-1}\psi_{\gamma_c(Reject)}) \\
\wedge & \mathbf{G}^{-1}(\psi_{\text{finish-Deliver}} \implies \mathbf{X}^{-1}\psi_{?\mathit{Deliver}}) \\
\wedge & \mathbf{G}^{-1}(\psi_{\text{finish-Reject}} \implies \mathbf{X}^{-1}\psi_{?\mathit{Reject}})
\end{array}$$

We detail next the formulas used above:

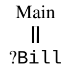
— The formula $\psi_{\gamma_c(Bill)}$ checks that the call guard of `Bill` is true:



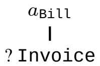
— The formula $\psi_{!\mathit{Bill}}$ checks that the call to `Bill` is not activated:



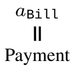
— The formula $\psi_{?\mathit{Bill}}$ checks that the function call to `Bill` is activated:



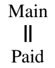
— The formula $\psi_{?Invoice}$ checks that the call to `Invoice` is activated:



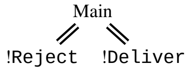
— The formula $\psi_{Payment}$ checks that the call to `Bill` has been returned:



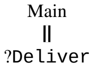
— The formula ψ_{Paid} checks that the call to `Bill` has been returned:



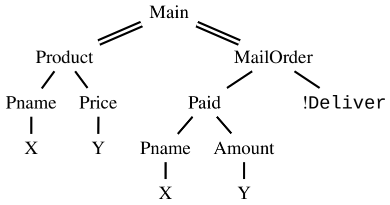
— The formula $\psi_{\text{!Deliver,!Reject}}$ checks that the calls to `Deliver` and `Reject` are not yet activated:



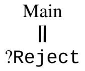
— The formula $\psi_{?Deliver}$ checks that the call to `Deliver` is activated:



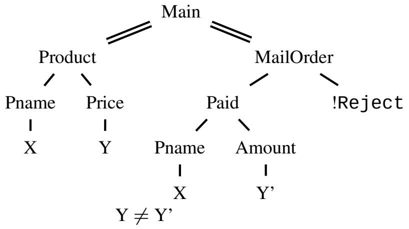
— The formula $\psi_{\gamma_c(Deliver)}$ checks that the call guard of `Deliver` is true:



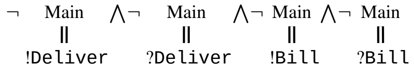
— The formula $\psi_{?Reject}$ checks that the call to `Reject` is activated:



— The formula $\psi_{\gamma_c(Reject)}$ checks that the call guard of `Reject` is true:



— The formula $\psi_{\text{finish-Deliver}}$ checks that `Deliver` has returned by checking that the function calls to `Deliver` and `Bill` no longer appear in the document:



— Finally, the formula $\psi_{\text{finish-Reject}}$ checks that `Reject` has returned by checking that the function calls to `Reject` and `Bill` no longer appear in the document:

$$\neg \begin{array}{c}\text{Main}\\ \| \\ \texttt{!Reject}\end{array} \bigwedge \neg \begin{array}{c}\text{Main}\\ \| \\ \texttt{?Reject}\end{array} \bigwedge \neg \begin{array}{c}\text{Main}\\ \| \\ \texttt{!Bill}\end{array} \bigwedge \neg \begin{array}{c}\text{Main}\\ \| \\ \texttt{?Bill}\end{array}$$

Checking workflow constraints

The following establishes the complexity of testing workflow constraints.

THEOREM 4.4. *For a fixed BAXML schema S and a fixed W where $W \in \{\mathcal{G}, \mathcal{A}, \mathcal{T}\}$, it is decidable in* PTIME *whether a given prerun ρ of S satisfies W.*

PROOF. Let $S|W$ be a workflow schema and $\rho = (I_i)$, $1 \leq i \leq n$ be a prerun of S. Note first that we can check that I_0 verifies the constraints of S and those imposed on initial instances by W in PTIME with respect to $|I_0|$. For $\gamma \in \mathcal{G}$, it is clear that one can further check, for each $i < n$, whether the transition from I_i to I_{i+1} satisfies γ in PTIME with respect to $|I_i| + |I_{i+1}|$. Consider an automaton $A = (Q, q_{init}, \delta, \Upsilon)$. To check that ρ satisfies A, we define by induction on i auxiliary relations R^i_q for each state $q \in Q$ as follows. For $i = 0$, all R_q are empty except $R_{q_{init}}$ that contains all valuations ν of $\overline{X}_{q_{init}}$ for which $I_0, \nu \models \Upsilon(q_{init})$. For $i > 0$, R^i_q contains all valuations ν of $\bar{X}_q$ for which there exists a sequence (q_j, ν_j), $j \leq i$, where $q_0 = q_{init}$, $q_i = q$, $\nu = \nu_i$, and for each $j < i$, ν_j is a valuation of $\overline{X}_{q_j}$, such that:

(1) $I_j, \nu_j \models \Upsilon(q_j)$,
(2) $\nu_j(\overline{X}_{q_j}) \cup \nu_{j+1}(\overline{X}_{q_{j+1}}) \models \delta(q_j, q_{j+1})$.

It is clear that for each i, the relations $\{R^{i+1}_q \mid q \in Q\}$ can be computed from I_{i+1} and $\{R^i_q \mid q \in Q\}$ in polynomial time. Moreover, the size of the relations R^{i+1}_q remains polynomial in the number of data values occurring in the entire prefix (I_j), $0 \leq j \leq i + 1$. Therefore, the set of relations $\{R^n_q \mid q \in Q\}$ can be constructed in time polynomial in $|\rho|$. Finally, ρ satisfies A iff some relation R^n_q is nonempty for some q.

Finally, consider $\mathcal{T}$. Let θ be a Past-Tree-LTL formula $\exists \overline{X} \psi(\overline{X})$. We must check that for some valuation ν of $\overline{X}$ to data values in ρ, ρ satisfies $\theta_\nu = \psi(\nu)$. Observe that θ_ν has no global variables. Let θ^0_ν be a Past-LTL propositional formula from which θ_ν is obtained by interpreting the propositions by Boolean pattern formulas. To each truth assignment of the propositions, one can assign a symbol. Let Σ be this set of symbols. There exists an automaton A_0 with alphabet Σ, that is equivalent to θ^0_ν. From A_0 it is straightforward to construct a tree-pattern automaton A_ν such that $S|\theta_\nu$ and $S|A_\nu$ have the same runs. Using the earlier result for automata, we can check that ρ satisfies A_ν in polynomial time. Moreover, it can be seen that the polynomial bound is independent of ν. Since there are polynomially many ν (for fixed ψ), it can be checked in PTIME whether ρ satisfies ψ. □

Remark 4.5. The complexity analysis in Theorem 4.4 assumes a fixed workflow schema. It is easily seen that the combined complexity (with respect to both prerun and schema) is upper-bounded by EXPTIME.

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 4.6. *(i) It is undecidable, given a BAXML schema S and a prerun ρ of S, whether ρ is blocking.*
(ii) It is undecidable, given a BAXML schema S with non-recursive DTD and a prerun ρ of S, whether ρ is blocking.

PROOF. (i) The undecidability is due to the external functions. We have to test whether there is some returned data that would be valid for the static constraints. This is undecidable because of the undecidability of the satisfiability of Boolean combinations of tree patterns under arbitrary DTDs [David 2008].

(ii) For each non-activated function call !f, it is sufficient to test it directly, and similarly for the return of an internal function call. Let ?f be an activated external function call. The problem of the possible return of ?f can be reduced to the satisfiability of a Boolean combination of patterns by an instance satisfying a non-recursive DTD, which is decidable [David 2008]. First, the DTD of the answer of the function f is rewritten to take into account the sibling trees of the function call ?f and the DTD of the schema. The rewritten DTD τ' ensures in particular that $(*)$ for a returned forest F, there exists a forest F' having the same multiset of the labels of roots as F and any tree of F' is isomorphic to a sibling of ?f. Intuitively, the construction of the Boolean combination of patterns is done by looking for patterns that can extend prefixes of patterns of the static constraints already mapped into the current instance. The extraction of the Boolean combination φ' from the static constraints is done as follows: Each pattern P is rewritten as a disjunction $\vee\varphi_{P,P'}(\nu)$, where P' is a prefix of P and ν a valuation of the variables of P'. A formula $\varphi_{P,P'}(\nu)$ is in the disjunction iff there is a mapping of $P'(\nu)$ in the instance I that can be extended to each node n of P not in P' but whose parent is in P', such that n can be mapped to $?\mathtt{f}$. The definition of $\varphi_{P,P'}(\nu)$ is the conjunction of subpatterns $[n]_P(\nu)$. A pattern $[n]_P$ is defined as follows:

— If the incoming edge to n is a child edge, then $[n]_P$ is the subtree rooted at n.
— If the incoming edge to n is a descendant edge, then $[n]_P$ is a root labeled with $*$ and its only subtree is the subtree rooted at n. The edge between the root and the subtree is a descendant edge.

If P and P' are equal then $\varphi_{P,P}(\nu)$ is set to *true*.

The formula φ' is satisfiable for reduced trees under τ' iff the function $?\mathtt{f}$ can return. □

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.

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 root is the empty prefix, and its state label is the empty instance. The state label for each node other than the root is the last instance in the prefix. For each non-root node ν, there is an edge labeled e from ν to node 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. The root has an outgoing edge to each node consisting of a prefix of length one, labeled by a distinguished event *init*. Thus, transitions from the root simply provide the initial instances of runs, and the infinite paths starting from children of the root 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 choices 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.

5.1. 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 2.5, 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.

Given workflow specifications W_1 and W_2, we denote by $W_1 \equiv W_2$ the fact that W_1 and W_2 have the same sets of runs.

THEOREM 5.1. *The workflow languages GAXML, AAXML and TAXML are incomparable relative to* $\hookrightarrow_{(id,id)}$.

We prove the theorem by a sequence of lemmas. The first two state that GAXML $\not\hookrightarrow_{(id,id)}$ {AAXML, TAXML} (by showing that there is a GAXML schema for which no AAXML or TAXML schema has the same set of runs). In both cases, we use the fact that, over data-free schemas (fixed vocabulary), the runs accepted by automata and by Past-Tree-LTL formulas are closed under equivalence with respect to homomorphism. (homomorphisms apply here just to the forests of the instances and ignore the mappings *eval*). Indeed, this follows from the fact that allowed transitions between instances depend in both cases only on the patterns satisfied by the instances, and satisfaction of patterns is preserved under homomorphism of data-free instances. Note that this is *not* the case for GAXML, as illustrated by the example constructed in the proof of the next lemma.

LEMMA 5.2. *GAXML* $\not\hookrightarrow_{(id,id)}$ *AAXML. In other words, there exists a GAXML schema* $S|\gamma$ *for which there is no AAXML schema* $S'|A$ *such that* $S|\gamma \;\equiv\; S'|A$.

PROOF. We describe a GAXML schema $S|\gamma$ for which no AAXML schema has the same set of runs. The DTD of S imposes that its initial instance consists of a tree of root r with five children labeled by function calls to some internal functions $\mathtt{f}_1, ..., \mathtt{f}_4$ and $\mathtt{end}$. The argument query of each $\mathtt{f}_i$ yields $\mathtt{f}$ for some internal function $\mathtt{f}$ and its return guard is false. The argument query of $\mathtt{f}$ produces some internal function $\mathtt{g}$ and its return guard is also false.

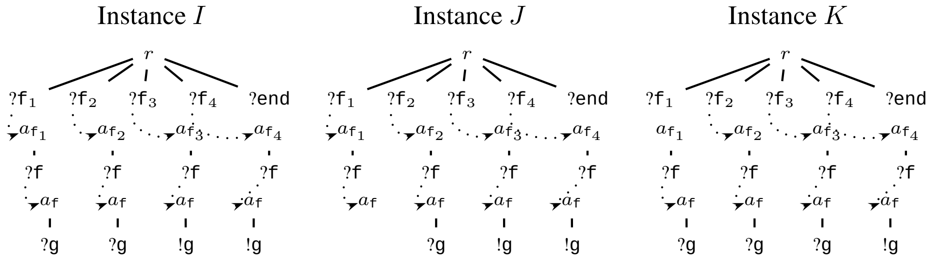


Fig. 10: Instances I, J and K.

Function g returns the empty message (its return guard is true). Function end has an empty argument query and a return guard that is false. In γ_c, all call guards are true except for g that is: end must not be active.

Consider a prerun ρ_0 of $S|\gamma$ resulting from the following transitions:

(1) call all functions $\texttt{f}_i$, $1 \leq i \leq 4$;
(2) call all functions f in the workspaces of the $\texttt{f}_i$;
(3) call 2 of the functions g in the workspaces of the functions f;
(4) call function end.

Clearly, this sequence of transitions is allowed by $S|\gamma$. Let I be the resulting instance. Now consider two transitions from I:

(i) return one of the two running calls to g, yielding instance J;
(ii) activate one of the two calls !g, yielding instance K.

Note that transition (i) is allowed by $S|\gamma$ whereas (ii) is not because the guard of g is false in I. Let ρ_J and ρ_K be the extensions of ρ_0 with transition (i) and (ii), respectively. Note that ρ_J and ρ_K are homomorphically equivalent. The instances I, J and K are represented in Figure 10.

Now suppose that there is an AAXML schema $S'|A$ equivalent to $S|\gamma$. Since ρ_J is a prerun of $S|\gamma$, it must also be a prerun of $S'|A$. Since runs satisfying AAXML schemas are closed under homomorphic equivalence, ρ_K must also be a prerun of $S'|A$. This contradicts the equivalence with $S|\gamma$, since ρ_K is not a prerun of $S|\gamma$.

Finally, note that it is necessary to have four initial functions $\texttt{f}_1, \ldots, \texttt{f}_4$, yielding four occurrences of g in I. Indeed, if there are only three initial functions (so three 3 g's in I) , it is easy to see that the instances K and J are no longer guaranteed to be homomorphically equivalent. □

Observe that the proof does not use relative patterns in guards.

LEMMA 5.3. *GAXML $\not\rightarrow_{(id,id)}$ TAXML. In other words, there exists a GAXML schema $S|\gamma$ for which there is no TAXML schema $S'|\theta$ such that $S|\gamma \;\equiv\; S'|\theta$.*

PROOF. This follows by a similar observation as above: the set of runs definable by a Past-Tree-LTL formula is closed under equivalence with respect to homomorphism (without data values). This is because the satisfaction of a Past-Tree-LTL formula by a prerun is determined by the patterns satisfied by each instance in the prerun, and homomorphic instances satisfy the same patterns. The details are straightforward and omitted. □

The next two lemmas state that GAXML cannot simulate AAXML or TAXML. In both cases, we use the fact that the history of the computation is not recorded in the current instance.

LEMMA 5.4. *AAXML $\not\rightarrow_{(id,id)}$ GAXML. In other words, there exists an AAXML schema $S|A$ for which there is no GAXML schema $S'|\gamma$ such that $S|A \;\equiv\; S'|\gamma$.*

PROOF. Consider the following AAXML schema $S|A$. The DTD of S enforces that the initial instance consists of one of the function calls !f or !g under the root, where f and g are non-continuous internal functions. There are no data values. A call to f returns !g and a call to g never returns (so all runs are blocking). The automaton A enforces that we start in a state q_{init} (with formula *true*), move to $q_{call\text{-}f}$ (with formula stating that ?f is a child of the root), move to q_{end} (with formula *true*). This imposes that if we start with f, we call f, receive !g, then call g and block; but if we start with g, we immediately block. Now suppose towards a contradiction that there exists a schema S' and a guard constraint γ so that $S'|\gamma \equiv S|A$. Observe that in the run starting from `f` under the root, we reach an instance I that consists only of `g` under the root and then `g` is called in I. Now use I as an initial instance. Then the guard of `g` allows calling `g` from I, a contradiction. □

LEMMA 5.5. *TAXML* $\not\hookrightarrow_{(id,id)}$ *GAXML. In other words, there exists a TAXML schema* $S|\theta$ *for which there is no GAXML schema* $S'|\gamma$ *such that* $S|\theta \;\equiv\; S'|\gamma$.

PROOF. The proof is the same as for AAXML $\not\hookrightarrow_{(id,id)}$ GAXML, where instead of the automaton A we use a constraint $\theta \in \mathfrak{T}$ stating that the initial instance has `!f` under the root. □

LEMMA 5.6. *TAXML* $\not\hookrightarrow_{(id,id)}$ *AAXML. In other words, there exists a TAXML schema* $S|\theta$ *for which there is no AAXML schema* $S'|A$ *such that* $S|\theta \;\equiv\; S'|A$.

PROOF. The proof is based on the fact that a Past-Tree-LTL formula can "remember" a data value even after it disappears from the instance, using an existentially quantified global variable, while this is not possible for an automaton (all parameters of a state must occur in the present instance). Specifically, consider a TAXML schema $S|\theta$ whose initial document consists of a single function call `!f` under root r. A call to `f` produces a workspace consisting of an external function call `!g` that returns a single data value. The function `f` returns a call to another external function `!h` that again returns a single data value. The Past-Tree-LTL formula θ imposes the following sequence of calls and returns:

(1) `f` is called
(2) `g` is called
(3) `g` returns a value u
(4) `f` returns `!h`
(5) `h` is called and returns the same value u returned in step (3).

Now suppose that there exists an $AAXML$ schema $S'|A$ describing the same sequence. The state of A after step (4) cannot have any parameters, since the current instance has no data value. Then A cannot impose that the data value returned in step (5) is the same as that in (3). Thus, no such automaton exists. □

The next lemma uses the fact that LTL is weaker than automata on finite words [Libkin 2004].

LEMMA 5.7. *AAXML* $\not\hookrightarrow_{(id,id)}$ *TAXML. In other words, there exists an AAXML schema* $S|A$ *for which there is no TAXML schema* $S'|\theta$ *such that* $S|A \;\equiv\; S'|\theta$.

PROOF. We use the following AAXML schema $S|A$. The DTD states that the root is r and it has two children, namely `!f` or ?`f` and `!g` or ?`g`. The function `f` is a continuous internal function that returns an empty answer. The function `g` never returns. From q_{init}, the automaton enforces that `f` is called, returns its answer, and is called again to get to a state q_{choice}. In that state, one can either return `f` and go back to q_{init} or call `g` and get to state q_{block}. Consider the four possible instances of S. We denote them by the symbols a (children of r are `!f`, `!g`), b (they are ?`f`, `!g`), c (they are ?`f`, ?`g`), and d (they are `!f`,?`g`). Observe that the set of preruns of $S|A$ is the prefix-closure L of the language $\{(ab)^{2n}c \mid n \geq 0\}$. Note that L cannot be expressed by FO on words because it is not counter free [Diekert and Gastin 2008], so it can neither be expressed by LTL [Libkin 2004]. Now suppose, towards a contradiction, that there exists a Past-Tree-LTL schema $S'|\theta$ equivalent to $S|A$. We show that we can construct from $S'|\theta$ an LTL formula φ that defines L. Apart from θ itself, the formula φ must capture the valid transitions among instances, as well as the DTD Δ of S'. Thus, φ is the conjunction of the following LTL formulas:

ψ_θ obtained from θ by replacing each pattern p by the disjunction of the symbols corresponding to the instances satisfying p (for example, for the pattern stating the existence of ?`f`, the disjunction is $b \vee d$), and replacing Past-LTL operators with LTL ones;

$\psi_\vdash$ is the conjunction of constraints on consecutive instances defining the transition relation $\vdash$ (for example, one such constraint is $\mathbf{G}(a \rightarrow \mathbf{X}(b \vee d))$);

ψ_Δ Note that Δ must allow instances a, b, c that appear in runs of $S|A$. Thus, Δ defines either $\{a, b, c\}$, the set of instances of $S'|\theta$, or $\{a, b, c, d\}$. If Δ defines $\{a, b, c\}$, then ψ_Δ is $\mathbf{G}(a \vee b \vee c)$. If Δ defines $\{a, b, c, d\}$, then ψ_Δ is *true*.

Let $\varphi = \psi_\theta \wedge \psi_\vdash \wedge \psi_\Delta$. It is easy to check that φ is an LTL formula defining L, contradiction. □

This concludes the proof of Theorem 5.1.

5.2. Comparison with projection views

Given the negative result of Theorem 5.1, 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 ($\Sigma_0 \subset \Sigma$) and $\mathcal{F}_0$ ($\mathcal{F}_0 \subset \mathcal{F}$) 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 $\mathcal{F}_0$. Note that, since the view used for the simulated schema is the identity, the visible tags and functions used in the simulation results are precisely those of that schema.

The projection[4] $\pi_{\Sigma_0,\mathcal{F}_0}([S])$ 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 $\mathcal{F}_0$ and their descendants. We also remove the workspaces whose corresponding function calls have been projected out. The projection of an event $!\mathtt{f}(F)$ is ϵ for $\mathtt{f} \notin \mathcal{F}_0$ and $!\mathtt{f}(\pi_{\Sigma_0,\mathcal{F}_0}(F))$ for $\mathtt{f} \in \mathcal{F}_0$, and similarly for $?\mathtt{f}(F)$. In addition, all projections preserve the special events $init$ and *block*. The projection view is defined in the same way for BAXML augmented with constraints (GAXML, AAXML, and TAXML).

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, i.e. the occurrence of two sibling nodes labeled $?\mathtt{f}$, for the same external function $\mathtt{f}$ (this can be enforced by the DTD). We denote these restrictions by AAXML^{sib} and TAXML^{sib}.

THEOREM 5.8. $\mathcal{W} \hookrightarrow_{(id,\pi)}$ BAXML *for* $\mathcal{W} \in \{\text{GAXML}, \text{TAXML}^{sib}, \text{AAXML}^{sib}\}$.

PROOF. We describe the three simulations needed to establish the result.

Simulation of GAXML by BAXML

We present the simulation in two stages: first, we demonstrate that the return guards can be removed from GAXML schema without losing expressiveness. Then, we demonstrate that a GAXML schema where all return guards are true can be simulated by a BAXML schema. We denote the set of GAXML schemas whose return guards are set to *true* by GAXML$^{\textit{no-ret}}$.

LEMMA 5.9. *GAXML* $\hookrightarrow_{(id,\pi)}$ *GAXML*$^{\text{no-ret}}$.

PROOF. We explain how we can remove the return guards of GAXML schemas.

Consider a GAXML schema $S|\gamma$. Due to Lemma 2.5 (Composition Lemma), and the fact that the set π of views is closed under composition, it is sufficient to show how to eliminate the return guards one function at a time.

Let $\mathtt{f}$ be an internal function of $S|\gamma$. Intuitively, we simulate the check of the return guard of a workspace of $?\mathtt{f}$ using a function call $!\mathtt{check\text{-}rg}_f$ in the same workspace, whose call guard checks the return guard of $\mathtt{f}$. We wish to ensure the following property, while maintaining the requirements of w-bisimulation:

(+) the call to $?\mathtt{f}$ can return only if the call to $!\mathtt{check\text{-}rg}_f$ has been activated in its workspace (signaling satisfaction of the return guard) and no other transition visible in the workspace occurred in the meantime.

Enforcing $(+)$ involves several subtleties, which we discuss in some detail in this first simulation proof. The same subtleties are addressed implicitly in the other simulations.

We explain how $(+)$ is enforced in several stages. We begin with a first attempt, that will have to be refined in order to satisfy the requirements of w-bisimulation.

Recall that, by definition, the answer of a call to $\mathtt{f}$ cannot be returned as long as the workspace of the call to $\mathtt{f}$ contains active function calls. Consider the following modification of the GAXML schema $S|\gamma$:

(i) the set of functions is augmented with an internal, non-continuous function $\mathtt{check\text{-}rg}_f$ with empty answer, whose call guard checks that the return guard of $\mathtt{f}$ holds, and that the workspace of the call to $\mathtt{f}$ contains no active function calls;

(ii) the argument query of $\mathtt{f}$ is modified so that its initial workspace contains a call to $!\mathtt{check\text{-}rg}_f$;

(iii) for every function $\mathtt{g}$, its call guard $\gamma_c(\mathtt{g})$ is replaced by $\gamma_c(\mathtt{g}) \wedge \alpha$ where α checks that, if $!\mathtt{g}$ occurs in a workspace of $\mathtt{f}$, then $!\mathtt{check\text{-}rg}_f$ also occurs in the same workspace (this can be done with relative patterns);

(iv) the return query of $\mathtt{f}$ is augmented with the rule

$$a_{\mathtt{f}}//!\mathtt{check\text{-}rg}_f \longrightarrow \{\textit{error}\}$$

[4] Recall that $[S]$ denotes the semantics of S, i.e. the workflow system it defines.

(v) the set of constraints of S is augmented to forbid the occurrence of *error*.
(vi) the return guard of f is set to *true*;

Let $S_1|\gamma_1$ be the resulting GAXML$^{\textit{no-ret}}$ schema. It easily seen that, whenever the answer of a call to f is returned in $S_1|\gamma_1$, the return guard of f in $S|\gamma$ is satisfied. Indeed, (ii) ensures that !check-rg$_f$ occurs initially in the workspace of the call, (iv) and (v) ensure that the answer cannot be returned before !check-rg$_f$ is activated, the call guard of check-rg$_f$ ensures that the return guard of f in $S|\gamma$ holds when check-rg$_f$ is activated, and (iii) together with the call guard of check-rg$_f$ ensure that no transition may occur in the workspace after check-rg$_f$ is activated.

While $S_1|\gamma_1$ seems to satisfy the intuition of the desired simulation, it is not quite satisfactory. Consider the view $V \in \pi$ for which the visible functions and tags are those of S, and consider the workflow systems $[S|\gamma]$ and $V([S_1|\gamma_1])$. We would like to have a w-bisimulation relation B from $[S|\gamma]$ to $V([S_1|\gamma_1])$. In particular, if $[S|\gamma]$ has no blocking states, neither should $V([S_1|\gamma_1])$. However, the above construction may yield blocking states in $[S_1|\gamma_1]$ (so also in $V([S_1|\gamma_1])$), even if no such states occur in $[S|\gamma]$. This is due to the fact that the activation of !check-rg$_f$ nondeterministically freezes the workspace in its current state. Although the return guard of f is satisfied at that point, the constraints of S may prohibit the instance resulting from the return, thus inhibiting it. This may result in a blocking state in $[S_1|\gamma_1]$, even if no such state occurs in $[S|\gamma]$.

To deal with the issue of blocking states, we must allow unblocking a workspace in which !check-rg$_f$ has been activated, and repeating the process. Note that we cannot simply make !check-rg$_f$ continuous, because the presence of !check-rg$_f$ prevents the return of the answer, by (iv). Instead, we can introduce an intermediate function, say rg-ok$_f$, that is returned by check-rg$_f$ and can in turn generate another call !check-rg$_f$. In more detail, let rg-ok$_f$ be an internal, non-continuous function, and modify check-rg$_f$ so that its answer returns the call !rg-ok$_f$. The call guard of !rg-ok$_f$ is *true* and its answer returns a call !check-rg$_f$. Let $S_2|\gamma_2$ be the resulting schema. It is clear that $S_2|\gamma_2$ prevents the undesired blocking encountered in $S_1|\gamma_1$.

However, we are not quite done, because the repeated trials yield in $V([S_2|\gamma_2])$ infinite sequences of silent transitions. These are due to infinite alternations of calls to !check-rg$_f$ and !rg-ok$_f$, without any intermediate visible function call or return. This violates the definition of w-bisimulation for $[S|\gamma]$ and $V([S_2|\gamma_2])$, since no such sequences exist in $[S|\gamma]$ (in fact $[S|\gamma]$ has no silent transitions at all). To circumvent this problem, we wish to ensure that some visible transition occurs between each return of the answer to ?check-rg$_f$ (yielding !rg-ok$_f$) and the next call to !rg-ok$_f$. Since attempts at returning the answer to f need only be made when no visible active calls exist in the workspace, it is sufficient to require the occurrence of at least one visible function call return. To detect such returns, we use a new auxiliary function return, and modify the answer queries of all visible functions so that every answer contains a call to !return. To allow visible functions to be activated following the activation of !check-rg$_f$, we remove the requirement imposed by (iii) above that their call guards require the presence of !check-rg$_f$. However, now we must ensure that the answer of f is not returned until !check-rg$_f$ is again activated, checking that the return guard of f still holds. This can be done by inhibiting the return of the answer of f while !return is present, similarly to (iv)-(v) above. In more detail, we modify $S_2|\gamma_2$ as follows:

(a) add the function return as an internal, non-continuous function returning the empty answer, and whose call guard requires the presence of !check-rg$_f$;
(b) modify the return queries of all visible functions so that their answer includes a call !return;
(c) restore the original guards of visible functions (undo (iii) above);
(d) modify the call guard of rg-ok$_f$ to require the presence of !return;
(e) augment the call guard of check-rg$_f$ to require the absence of !return or ?return.
(f) add to the answer query of f the rule:

$$a_{\texttt{f}}//!\texttt{return}_f \longrightarrow \{\textit{error}\}$$

Let the resulting schema be $S_3|\gamma_3$. Note that, due to (b), the new function return affects the entire instance, not just the workspaces of f. When it occurs outside a workspace of f, its call guard cannot hold, so the call is never activated. Its presence is however harmless because it does not cause transitions and is not visible in $V([S_3|\gamma_3])$.

We claim that $S|\gamma$ and $V(S_3|\gamma_3)$ are now w-bisimilar. More precisely, let B be the relation from the nodes of $[S|\gamma]$ to those of $V([S_3|\gamma_3])$ defined as follows. Recall that both $[S|\gamma]$ and $V([S_3|\gamma_3])$ have as root the empty run, which we denote $\rho_\emptyset$. The relation B is the smallest relation satisfying the following:

— $B(\rho_\emptyset, \rho_\emptyset)$
— if $B(s_1, q_1)$ and $s_1 \xrightarrow{e} s_2$, $q_1 \xrightarrow{e} q_2$ for some visible event e, then $B(s_2, q_2)$.

From the above discussion it follows that B is a w-bisimulation relation. This completes the proof. □

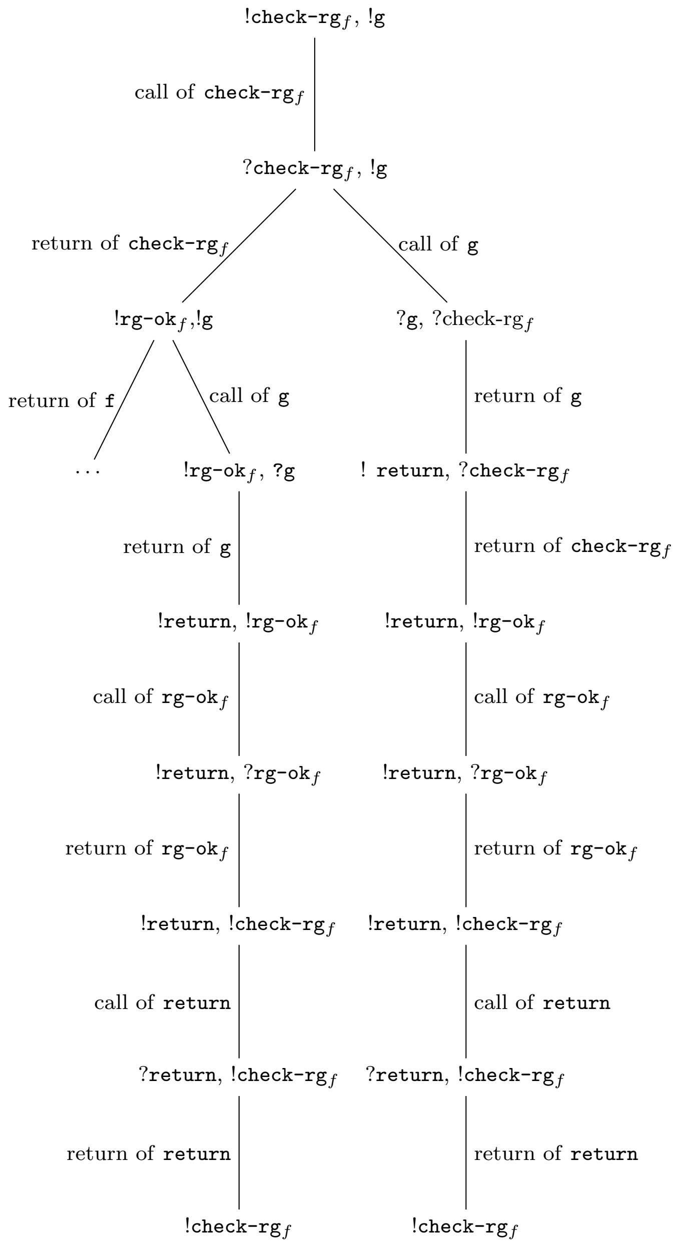


Fig. 11: Tree illustrating some of the possible actions in the simulation of the return of the function f in Example 5.10

Example 5.10. To illustrate the construction in the previous proof, we consider the following simple example. Let $S|\gamma$ be a GAXML schema, and suppose an instance I is reached that contains an activated call to a function f. Suppose the workspace of f consists of just one unactivated function call to a visible function, !g. We assume that the return guard of f and the call guard of g are satisfied by I. Figure 11 gives an overview of the possible sequences of function calls and returns in the simulation of $S|\gamma$ by $S_3|\gamma_3$.

We next show that GAXML without return guards can be simulated by BAXML.

LEMMA 5.11. *GAXML*$^{\text{no-ret}} \hookrightarrow_{(id,\pi)}$ *BAXML.*

PROOF. Let $S|\gamma$ be a GAXML$^{\textit{no-ret}}$ schema. We construct a BAXML schema S' that simulates $S|\gamma$. Intuitively, we check the guard of f by adding to the argument query of f additional rules that check satisfaction of each pattern of $\gamma_c(\texttt{f})$ and insert a corresponding tag in the workspace, signaling satisfaction of the pattern. Specifically, for each pattern P of $\gamma_c(\texttt{f})$, we add to the argument query of $\texttt{f}$ a rule $P \rightarrow \{sat_P\}$ where sat_P is a new tag. Note that, if P is a relative pattern, *self* is mapped to the same node when it is viewed as the body of a relative query. Finally, the DTD of the workspace is modified to allow only subsets of tags sat_P corresponding to truth assignments satisfying $\gamma_c(\texttt{f})$. This ensures that $!\texttt{f}$ can only be activated if $\gamma_c(\texttt{f})$ is satisfied. Remark that this construction works only for internal functions, as external function calls do not produce a workspace. To deal with external functions, the schema is first modified to ensure that every new occurrence of an external call $!\texttt{f}$ is accompanied by a sibling $!\texttt{lock}_f$. This is done using the DTDs (including those of answers to external functions), as well as by modifying the answer queries of internal functions by adding to every occurrence of $!\texttt{f}$ a sibling $!\texttt{lock}_f$.

The function $!\texttt{lock}_f$ is internal, non-continuous, and returns the empty answer. It has several roles:

— checking satisfaction of the guard of $\texttt{f}$; this is done as above, using the workspace of $\texttt{lock}_f$;
— checking that the static constraints *would* be satisfied after the activation of $!\texttt{f}$. This is done by rewriting the constraints in order to allow mapping $?\texttt{f}$ to $?\texttt{f}$ or to $?\texttt{lock}_f$ and $!\texttt{f}$ to $!\texttt{lock}_f$.

Static constraints require that $!\texttt{f}$ can only be activated if it has a sibling $?\texttt{lock}_f$, ensuring that its guard and constraints are true. In addition, $?\texttt{lock}_f$ acts as a lock disallowing any action other than the activation of the sibling $!\texttt{f}$. Specifically, we must prevent the following actions as long as $?\texttt{lock}_f$ is present:

— activation of another call $!\texttt{lock}_g$ for an external function $\texttt{g}$; this is prevented by having the call guard of each function $\texttt{lock}_g$ prohibit the existence of any other active call $?\texttt{lock}_h$ in the instance.
— activation of an internal function; to prevent this, we add a new, internal, non-continous function $\texttt{activated}$ and modify the argument queries of all internal functions in order to force the inclusion of a call $!\texttt{activated}$ in their answer. A constraint prohibits the simultaneous occurrence of $!\texttt{activated}$ and $?\texttt{lock}_f$ in the instance. The function $\texttt{activated}$ returns the empty answer.
— return of a function call; similarly to the proof of Lemma 5.9, we add a new internal, non-continuous function $\texttt{return}$ and modify the return queries of internal functions and the return DTD's of external functions so that their answers contain a call $!\texttt{return}$. A constraint prevents $?\texttt{lock}_f$ and $!\texttt{return}$ from occurring simultaneously. The function $\texttt{return}$ returns the empty answer.

Let S' be the resulting BAXML schema. Let $V \in \pi$ be the projection view for which the visible tags and events are those of S (and recall that init and *block*, are always visible). As in the proof of Lemma 5.9, let B be the smallest relation from the nodes of $[S|\gamma]$ to those of $V([S'])$ satisfying the following:

— $B(\rho_\emptyset, \rho_\emptyset)$
— if $B(s_1, q_1)$ and $s_1 \xrightarrow{e} s_2$, $q_1 \xrightarrow{e} q_2$ for some visible event[5] e, then $B(s_2, q_2)$.

A straightforward case analysis shows that B is a w-bisimulation relation from $[S|\gamma]$ to $V([S'])$. The only non-trivial aspect of the simulation concerns the functions $\texttt{lock}_f$. It is critical to note that every activation of $!\texttt{lock}_f$ leads to a successful call to $!\texttt{f}$ (so a visible event). This ensures that no extraneous blocking occurs in S', and also that there are no infinite chains of silent transitions. Thus, B is indeed a w-bisimulation. □

In summary, we have shown that

$$\textit{GAXML} \hookrightarrow_{(id,\pi)} \textit{GAXML}^{\textit{no-ret}}$$

$$\text{and } \textit{GAXML}^{\textit{no-ret}} \hookrightarrow_{(id,\pi)} \textit{BAXML}$$

By Lemma 2.6 it follows that GAXML $\hookrightarrow_{(id,\pi)}$ BAXML. Since this is the first application of the lemma, we explain it in detail. The lemma is applied with $\mathcal{V}_1 = \mathcal{V}_2 = \textit{id}$ and $\mathcal{V} = \mathcal{V}_3 = \pi$. Since $\pi = id \circ \pi$ we have that GAXML $\hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_2 \circ \mathcal{V})}$ GAXML$^{\textit{no-ret}}$ and GAXML$^{\textit{no-ret}}$ $\hookrightarrow_{(\mathcal{V}_2, \mathcal{V}_3)}$ BAXML. By Lemma 2.6, GAXML $\hookrightarrow_{(\mathcal{V}_1, \mathcal{V}_3 \circ \mathcal{V})}$ BAXML. Since $\pi \circ \pi = \pi$ it follows that GAXML $\hookrightarrow_{(id,\pi)}$ BAXML.

Simulation of AAXML^{sib} by BAXML

Let $S|A$ be an AAXML^{sib} schema with functions $\mathcal{F}_0$ and tags Σ_0. We outline the construction of a GAXML schema $S'|\gamma$ that simulates $S|A$ relative to projection views. Since GAXML can be simulated by BAXML relative to projection views, and since projection is coarser than the identity on GAXML, Lemma 2.6 implies that AAXML^{sib} can be simulated by BAXML.

[5] Recall that *init* and *block* are always visible events.

Without loss of generality, we can assume that the static constraints of S consist just of a DTD. Indeed, the data constraints can be easily pushed into the pattern automaton A. As described in the proof of Theorem 4.4, the satisfaction of an automaton A by a prerun can be checked incrementally by maintaining the states of the automaton reachable in the prerun, together with the valuations of their parameters. The simulation by a GAXML schema essentially implements the same incremental check. Thus, $S'|\gamma$ must alternate the simulation of events of $S|A$ (function calls and returns) with validity checks and updates of the state and valuation information of A. The simulation is quite intricate and we outline the main points, providing intuition on the more subtle aspects.

The representation and maintenance of the state and valuation information for A is straightforward. We use a subtree with root *states*, and one child `!q` for each state q of A. Valuations of $\bar{X}_q$ are kept in adjacent subtrees, each with root label V_q. The current valuations are marked by a function `!current` (internal, noncontinuous, with empty answer). An evaluation of `!q` returns a new set of valuations, also subtrees with root V_q, but now marked with another function `!new`. The update is completed by having the functions `!current` vanish and the functions `!new` turn into `!current`. One update round is controlled by a function `update` whose activation enables the update and blocks all transitions not involved in the update. Other locks ensure that `update` can be activated only when the simulation of one transition of S is completed. We can also enforce that the update round is performed only once between transitions.

The main difficulty in the simulation concerns the function calls and returns, and their timing relative to the update round outlined above. Specifically, the following raise technically intricate points:

(i) ensuring that validity of a function call or return is checked for each event (in particular, this requires preventing multiple transitions skipping intermediate validity checks and state/valuation updates)

(ii) checking validity of a candidate event of S with respect to the DTD and A without actually carrying out the event (in particular, one must prevent infinite branches of ϵ-transitions caused by unsuccessful guesses of the next valid event)

The sequencing needed for (i) and (ii) is enforced by a locking mechanism implemented by auxiliary functions. Before outlining the main aspects of the simulation, we make some useful technical remarks.

Valid automata transitions vs. static constraints Given the current state/valuation information for A and a *next* instance I of S, validity with respect to A of the transition to I can be expressed in S' by a formula φ_{next}. The formula φ_{next} is the disjunction $\vee_{q,q'}\ \psi_{\mathit{next}}(q,q')$, where q and q' are states of A, and $\psi_{\mathit{next}}(q,q')$ checks that q is a current state, the formula $\Upsilon(q')$ holds, and the equality constraints between some valuation of $\bar{X}_q$ and a possible next valuation of $\bar{X}_{q'}$ provided by $\Upsilon(q')$ are satisfied. Note that φ_{next} is not directly expressible as a static constraint in S', because these are Boolean combination of independent patterns, whereas φ_{next} uses parameterized patterns sharing free variables. To overcome this gap, some pre-processing is needed for each transition. Specifically, for a formula φ_{next} with free variables $\overline{X}$, candidate valuations for $\overline{X}$ are generated and the patterns in φ_{next} are augmented so that $\overline{X}$ is bound in all patterns to the same valuation. The generation of the candidate valuations depends on the action leading to the transition (we omit the details). This reduces evaluation of φ_{next} to evaluation of a Boolean combination of independent patterns, so a static constraint of S'. In the following, we will use for simplicity φ_{next} as a static constraint, bearing in mind that its evaluation requires the above pre-processing phase. Parameterized queries used in the automaton A yield another difficulty for the initial state. To ensure that the initial document satisfies the parametrized query of the initial state, we assume that there is only one valuation of the initial state represented in the initial document. This way the parametrized query can be simulated by a Boolean combination of patterns. The other valuations are built at the beginning of the simulation by the activation and the return of a function call `!init-valuation`.

Rewriting patterns The patterns used in $S|A$ have to be rewritten when used in $S'|\gamma$. Indeed, since an instance I' of S' contains the corresponding instance I in $S|A$, a pattern can be satisfied in I' and not in I. The main problem is due to descendant branches and the wildcard used in patterns. To resolve this, each tag in Σ_0 used in I' is adorned with a child labeled *real*. The patterns are rewritten using these markings, to ensure that each pattern of $S|A$ used in $S'|\gamma$ is mapped to nodes in I rather than to hidden nodes used in the simulation.

Rewriting queries The simulation introduces new data values in the trees. These data values can be matched by patterns in the queries, such as $q = *//\$x$. To avoid this, we first ensure by static constraints that each node labeled by a tag appearing in the projected trees has a child labeled *real*, as explained previously. Queries are rewritten in order to access only data values accessible using nodes having a child labeled *real*.

Extending GAXML with global return guards In our simulation, we allow return guards that can check a global property of the instance. This is an extension of GAXML, since in GAXML return guards of function calls are only able to check properties of the workspace. In our context, we can simulate global return guards. This is done by adding to the workspace of each function f using a global return guard $\gamma_r(\texttt{f})$ a function $\texttt{check-return-guard}_f$. The call guard of this function is $\gamma_r(\texttt{f})$. The new local return guard of f simply checks that $\texttt{check-return-guard}_f$ has returned. This works in the context of our simulation because we only use it on reachable instances I of $S|\gamma$ in which satisfaction of $\gamma_r(\texttt{f})$ implies that the return of the corresponding call to f leads to the only valid transition. Note that otherwise, a reevaluation of check-return-guard would have to be done after each other valid transition by using a mechanism like in the proof of GAXML without return guard.

We next outline the simulation of the events of $S|A$, making use of the above observations. In all cases, the simulation involves the following steps:

(1) Acquire a lock for a function call or return. The lock initiates an *attempt* to carry out the associated event.
(2) Check that the event corresponding to the lock would result in a valid transition of $S|A$.
(3) In the affirmative, the locked event is carried out and the lock released. Otherwise, the lock is also released, but in a manner that prevents another locking attempt before a valid event occurs. This prevents infinite branches of ϵ-transitions.

We now describe the specific simulation used for the activation of a function call, the return of an internal function call, and the return of an external function call.

Activation of a function call The activation of an internal function !f is controlled using a sibling function $!\texttt{lock}_f$. As described above, this has a dual role: it acts as a lock, and it checks whether the activation of !f *would* result in a transition allowed by the automaton. If so, it returns a function call !activate-f. Otherwise, it returns !notactivate-f. The call !f cannot be activated unless !activate-f occurs as a sibling. The functions $!\texttt{lock}_f$ and !activate-f also prevent other transitions from occurring during the attempt to activate !f. To this end, one can guarantee that there is at most one node labeled $?\texttt{lock}_f$, (for some f) in an instance, i.e. at most one lock. This is enforced by the guard of $\texttt{lock}_f$. Moreover, no active function call can return its answer while $?\texttt{lock}_f$, !activate-f, or ?activate-f occur. As described in the proof of Lemma 5.11 , it is easy to ensure that every occurrence of !f is always accompanied by a sibling $!\texttt{lock}_f$ following each visible transition.

To ensure that !f is activated whenever !activate-f is activated, the guard of activate-f ensures that this function cannot be called while it still has a sibling !f. The function call !notactivate-f ensures that $!\texttt{lock}_f$ cannot be called more than once between two valid transitions. It is activated during the maintenance phase and returns $!\texttt{lock}_f$ (needed for the next attempt to call !f, following another transition). The constraints impose that activate-f handshakes with the lock for the maintenance of the states and valuations.

Figure 12 summarizes the possible sequences of activations in the simulation of an internal call to f. The role of the function $\texttt{w}_{f,a}$ will be explained shortly. The nodes represent the functions that occur as siblings of the node labeled ?f or !f. . The possible sequences for an external call are the same except the function $\texttt{w}_{f,a}$ is replaced by $\texttt{certificate}_{f,a}$.

Return of an internal function call We describe the simulation in several stages. The basic locking mechanism is simple. The lock initiating an attempted return of a function call ?f is implemented using a function $!\texttt{lock}_w$ present in the workspace. If the call return to ?f would result in a valid transition, the lock is released and the result is returned. Otherwise, the lock is released and another function !wait is activated in order to inhibit any locking attempt until another transition has been successfully completed.

Checking validity of the call return is much more complex. It is carried out using the workspace of an auxiliary function $\texttt{check}_{f,a}$ that is a sibling to ?f (here a is the tag of the parent of ?f, needed to check the DTD). A difficulty is to make sure the activated occurrence of $\texttt{check}_{f,a}$ is indeed a sibling of the call ?f whose workspace is locked (recall that patterns cannot detect the link between a call and its workspace). Assume for the moment that this is achieved. Then $\texttt{check}_{f,a}$ works as follows. First, it generates in its workspace a copy of its sibling subtrees, (these are "almost" isomorphic copies of the originals, keeping sufficient information for checking validity, see below). This copy is initiated by the activation of copy-sibling appearing in the workspace of $\texttt{check}_{f,a}$. Next, it generates in the same workspace the answer to the locked call ?f. In the following stage, four functions are used to test satisfaction or violation of the DTD (ok-dtd and notok-dtd) and the automaton constraint (ok-A and notok-A) by the result. Specifically, for the first two the test is done using the DTD of S' and for the last two using their guards. To test satisfaction of the automaton constraint using guards, the formula φ_{next} has to be rewritten into a disjunction of formulas, each of which decomposes the patterns into a part that applies to the workspace of $\texttt{check}_{f,a}$ (mimicking the subtree rooted at the parent of the call $?\texttt{check}_{f,a}$, labeled a) and another to the rest of the instance. If the result is positive (the transition is valid) then a flag *ok-return* is turned on in the workspace of ?f. The guards and constraints

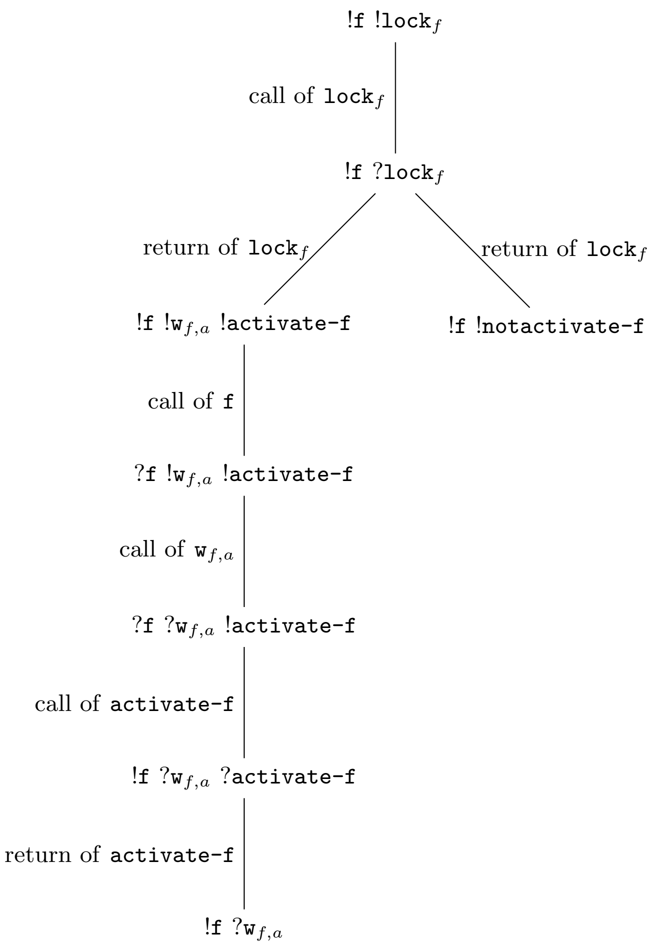


Fig. 12: Some of the actions for the simulation of the activation of the call to f

then force the answer to the call ?f to be returned, and ?check$_{f,a}$ returns the empty answer. If the result is negative, the function !wait is activated in the workspace of ?f (see above), and These functions are used to allow a new check of this function after a valid transition as detailed be.

We next explain how to generate !check$_{f,a}$ as a sibling of the call ?f whose workspace is locked. The process starts at the time when !f is activated. We ensure that each function call !f has as a sibling a call !w$_{f,a}$ (where a is the tag of the parent of the function call). When the call to !f is made, its workspace includes a function !init that uniquely marks the most recent function call (and later vanishes). Additionally, a new identifier α is generated in the workspace of ?f (more on this in the next paragraph). Then the function !w$_{f,a}$ is called and copies the identifier α from the workspace of ?f marked by !init. Note that the only function call !w$_{f,a}$ without a sibling !f is the sibling of the most recently activated call ?f. Once the simulation of the call to !f is completed, !init vanishes but the workspaces of ?f and ?w$_{f,a}$ remain linked by the identifier α. When the return of the call ?f is simulated, the call ?w$_{f,a}$ sharing the same id α with the workspace of ?f returns as answer the desired function call !check$_{f,a}$. If due to a lock the return of f is disallowed, the call to !w$_{f,a}$ has to be activated again. The function check$_{f,a}$ returns the function calls !w$_{f,a}$ and !reinitialize. The second function ensures that its sibling !f has as sibling !w$_{f,a}$ after the reinitialization.

The identifier α in the previous paragraph can easily be generated by an external function that returns a new value. If one wishes to avoid using external functions in the simulation, the identifier can be represented by a chain of calls to two internal functions, encoding the binary representation of an integer. The bookkeeping is more complicated in this case, since comparing identifiers is no longer an atomic operation. In particular, identifiers have to be destroyed and reconstructed (details omitted). Moreover, the identifiers have to be refreshed after each valid transition to ensure that the size of each instance of the simulation remains polynomial in the size of the current instance.

Recall that one of the roles of !$\texttt{check}_{f,a}$ is to copy the relevant sibling subtrees of ?f. We explain briefly how this is done. We enforce that each tag of Σ_0 has a child function call !copy-to. As remarked earlier, the copy performed loses some information. The loss concerns the exact number of sibling calls ?g to an internal function g. Indeed, it is not possible to fully replicate this information because of the limitations of patterns. Fortunately, multiple occurrences of sibling calls to the same function are not relevant when they occur as internal nodes in sibling subtrees of ?f. Thus, only one representative of such calls is copied. This does not affect the simulation, since trees with activated function calls cannot be merged, and patterns cannot count such occurrences. For calls ?g occurring as siblings of ?f, their number is relevant to satisfaction of the DTD after the call return, but only up to the maximum integer used in the DTD of S. The number of occurrences up to this maximum can be signaled using additional function calls whose activation is constrained by the DTD of S'. For example, the DTD may stipulate that a function !$\texttt{eq}_{(?\texttt{g}=m)}$ may be activated iff the number of occurrences of siblings ?g is m.

Copying a tree is done by mutually recursive calls between functions residing in the source tree (copy-to, in-progress, copy-values-to, done-to) and in the target tree (copy-from, copy-values-from, done-from). The copy is done in a depth-first manner. The copy-to indicates the parent node to copy. The function call !copy-from copies this node with child labeled !copy-from. The function call !in-progress indicates that copying is in progress for the subtrees of the parent node of this call. The function call copy-values-to indicates that the function calls and the sibling values of this function have to be copied. It implies that the subtrees are entirely copied, which is signaled by the function !done. The copy of the function calls is tricky, since copying the activated function calls has to be done before the others (to guarantee that partially copied subtrees are not merged). The function calls !done-from and !done-to are reinitialized to !copy-to after each valid transition.

Figure 13 summarizes the tree of actions done to check the return of a call. At each node, we represent the function calls siblings of the call ?f, the function calls in the workspace of ?f and the function calls in the workspace of $\texttt{check}_{f,a}$ when it is in the simulation.

Return of an external function call This is the most subtle part of the simulation. Observe first that it is not possible to take a lock using a marker returned by an external function call ?f, because two calls to ?f at different locations in the document may return exactly the same forest and be indistinguishable by the constraints of the GAXML schema. Moreover, it is not possible to take a lock prior to the return of ?f, because one cannot know if ?f can return an answer satisfying the constraints (recall that this is undecidable, see proof of Theorem 4.6). If a lock is taken when ?f cannot return, this leads to a blocking run in an instance of $S'|\gamma$ whose projection in $S|A$ is not blocking, which violates the definition of simulation. Instead, the idea of our simulation is to use, for every call ?f to an external function, an associated sibling call to an internal function $\texttt{certificate}_{f,a}$ such that:

(i) if ?f may return, then ?$\texttt{certificate}_{f,a}$ *may* return a flag !$\texttt{return}_f$. The function !$\texttt{return}_f$ compels ?f to return and also acts as a lock preventing other transitions until the next cleaning stage.

(ii) the call ?$\texttt{certificate}_{f,a}$ may remain activated until the next cleaning stage, in which case ?f is not allowed to return. During the cleaning stage, the call ?$\texttt{certificate}_{f,a}$ returns and is reactivated.

Note that, even if ?f can return, ?$\texttt{certificate}_{f,a}$ does not necessarily return, unless the return of ?f is the only possible next transition. Otherwise, the cleaning stage may be reached without a return of ?$\texttt{certificate}_{f,a}$ or ?f, by simulating some other transition. If ?$\texttt{certificate}_{f,a}$ does not return and the cleaning stage is not reached, then the run is blocking, both in $S|A$ and in $S'|\gamma$.

We next elaborate on (i). To mimick ?f, the function $\texttt{certificate}_{f,a}$ uses in its workspace an external control function $\texttt{fake}_f$. The workspace also contains additional information so that ?$\texttt{fake}_f$ may return in the context of the workspace iff ?f may return in the context of its original location. Specifically, the workspace contains a copy of the sibling subtrees of ?f (this is done as in the previous simulation). In addition, it contains information on the evaluation of the patterns in $\varphi_{\textit{next}}$ on the portion of the current instance excluding the siblings of ?f. The partial evaluations of the patterns together with the siblings allow expressing within the workspace constraints on the return of ?$\texttt{certificate}_{f,a}$ that are equivalent to those on the return of ?f (the DTD and valid transition in A). This ensures that ?$\texttt{fake}_f$ may return iff ?f may return. If ?$\texttt{fake}_f$ returns, then ?$\texttt{certificate}_{f,a}$ returns the flag !$\texttt{return}_f$ as desired. To prevent multiple returns to ?$\texttt{fake}_f$ at different locations in the document, the answer to ?$\texttt{fake}_f$ contains a flag !return-fake-f that is not allowed to appear twice in the document. To ensure this, the workspace of ?$\texttt{certificate}_{f,a}$ also contains a unique id (generated by an external function). A constraint forbids two occurrences of !return-fake-f with distinct workspace id's. Note that the id technique could not be used to implement directly a lock for the return of ?f, because such an id could not be erased from the instance and this could lead to faulty simulations. Indeed, the id's could inhibit merging of subtrees whose projections would otherwise be merged.

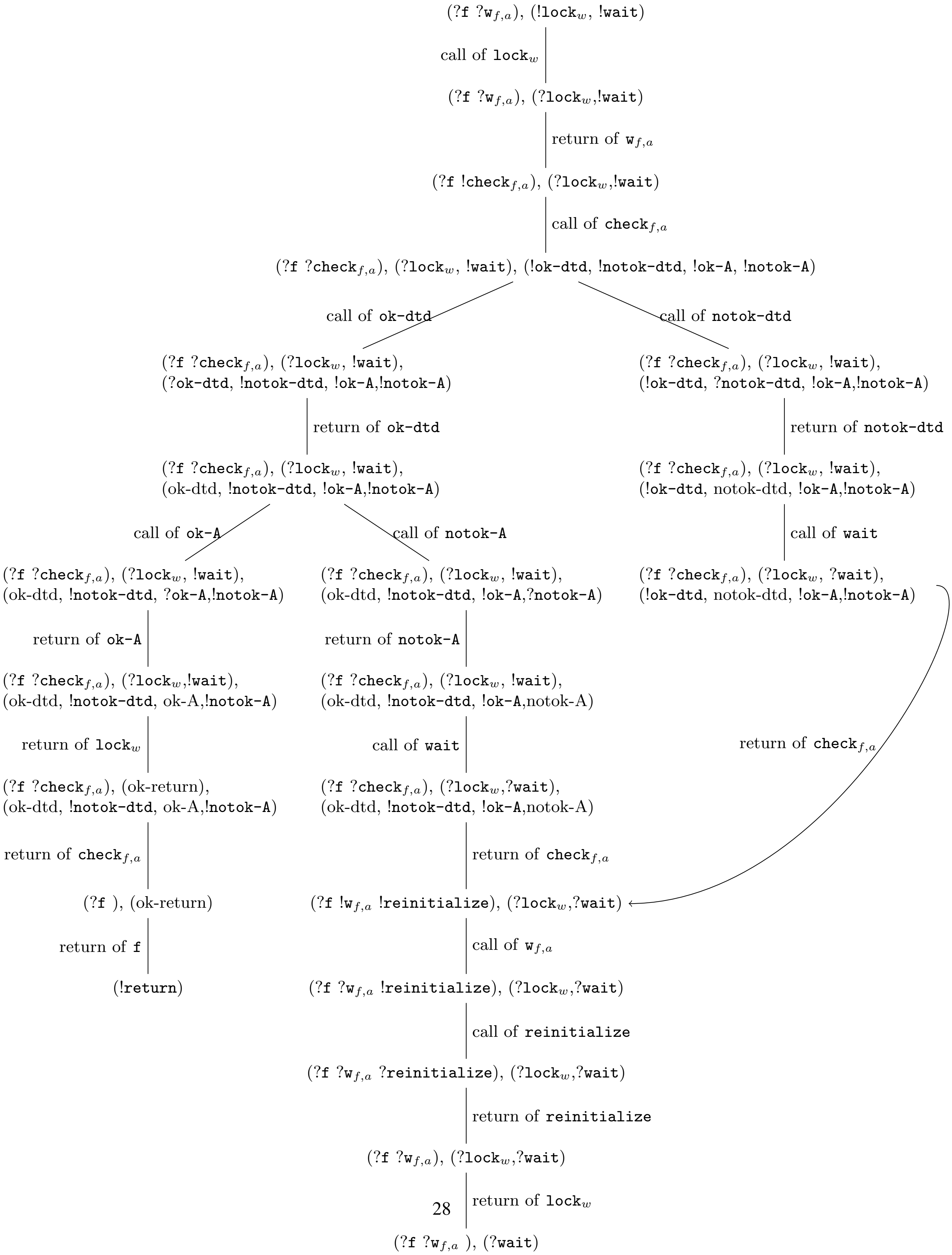

(?f ?$w_{f,a}$), (!$lock_w$, !wait)
call of $lock_w$
(?f ?$w_{f,a}$), (?$lock_w$,!wait)
return of $w_{f,a}$
(?f !$check_{f,a}$), (?$lock_w$,!wait)
call of $check_{f,a}$
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (!ok-dtd, !notok-dtd, !ok-A, !notok-A)
call of ok-dtd
call of notok-dtd
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (?ok-dtd, !notok-dtd, !ok-A,!notok-A)
return of ok-dtd
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (ok-dtd, !notok-dtd, !ok-A,!notok-A)
call of ok-A
call of notok-A
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (ok-dtd, !notok-dtd, ?ok-A,!notok-A)
return of ok-A
(?f ?$check_{f,a}$), (?$lock_w$,!wait), (ok-dtd, !notok-dtd, ok-A,!notok-A)
return of $lock_w$
(?f ?$check_{f,a}$), (ok-return), (ok-dtd, !notok-dtd, ok-A,!notok-A)
return of $check_{f,a}$
(?f), (ok-return)
return of f
(!return)
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (ok-dtd, !notok-dtd, !ok-A,?notok-A)
return of notok-A
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (ok-dtd, !notok-dtd, !ok-A,notok-A)
call of wait
(?f ?$check_{f,a}$), (?$lock_w$,?wait), (ok-dtd, !notok-dtd, !ok-A,notok-A)
return of $check_{f,a}$
(?f !$w_{f,a}$!reinitialize), (?$lock_w$,?wait)
call of $w_{f,a}$
(?f ?$w_{f,a}$!reinitialize), (?$lock_w$,?wait)
call of reinitialize
(?f ?$w_{f,a}$?reinitialize), (?$lock_w$,?wait)
return of reinitialize
(?f ?$w_{f,a}$), (?$lock_w$,?wait)
return of $lock_w$
(?f ?$w_{f,a}$), (?wait)
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (!ok-dtd, ?notok-dtd, !ok-A,!notok-A)
return of notok-dtd
(?f ?$check_{f,a}$), (?$lock_w$, !wait), (!ok-dtd, notok-dtd, !ok-A,!notok-A)
call of wait
(?f ?$check_{f,a}$), (?$lock_w$, ?wait), (!ok-dtd, notok-dtd, !ok-A,!notok-A)
return of $check_{f,a}$

Fig. 13: Some of the actions in the simulation of the return of the call to the internal function f

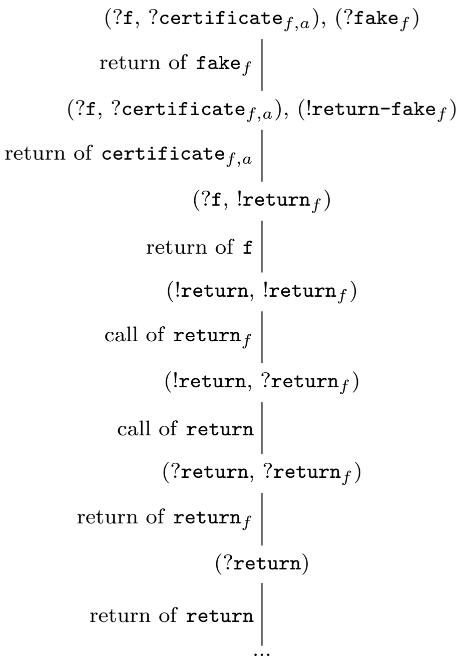


Fig. 14: Some of the actions for the simulation of the return of the call of the external function f

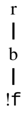
Finally, if ?certificate$_{f,a}$ does not return during the current round, its workspace is reconstructed during the cleaning stage in order to reflect changes in the instance.

Figure 14 summarizes the tree of actions performed to check the return of an external call. At each node, we represent the function calls occurring as sibling of the call ?f, then the function calls in the workspace of certificate$_{f,a}$ when it exists.

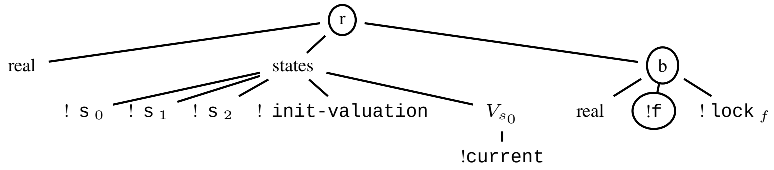
Example 5.12. We illustrate the main elements of the above simulation using a simple example. Consider the following AAXML schema $S|A$. Its static constraints consist of the following DTD:

$$\begin{aligned} r &\longrightarrow |b| = 1 \wedge |c| = 1 \\ b &\longrightarrow |a| = 1 \vee |!\mathtt{f}| = 1 \vee |?\mathtt{f}| = 1 \end{aligned}$$

S has one internal function f, whose argument and return queries always produce a single node labeled a. The automaton A has three states, s_0, s_1, s_2, with no associated variables. The initial state is s_0 and there are transitions from s_0 to s_1 and from s_1 to s_2. The formula associated with the initial state s_0 checks for the presence of !f , the formula for s_1 checks for the presence of ?f and the formula for s_2 checks for the absence of ?f. Thus, the only possible sequence of events is the activation of !f and the return of ?f. We describe how the two transitions are simulated by the GAXML schema $S'|\gamma$ with global return guards constructed in the proof. The initial AAXML instance is the following:

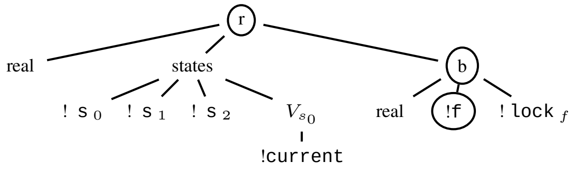


The shape of the initial document for $S|A$ is ensured by a pattern associated with s_0 (having the same tree representation as the instance above). The corresponding initial instance for the GAXML schema $S'|\gamma$ is the following: This is enforced by the static constraints of S'. For readability, we omit in figures the function calls copy-to introduced in the full proof to facilitate copying trees in the simulation (they should appear under every visible node). Also, all visible nodes are circled.

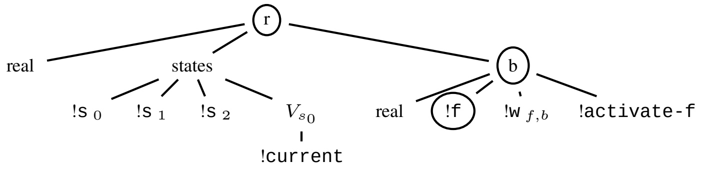


The simulation consists of the following steps.

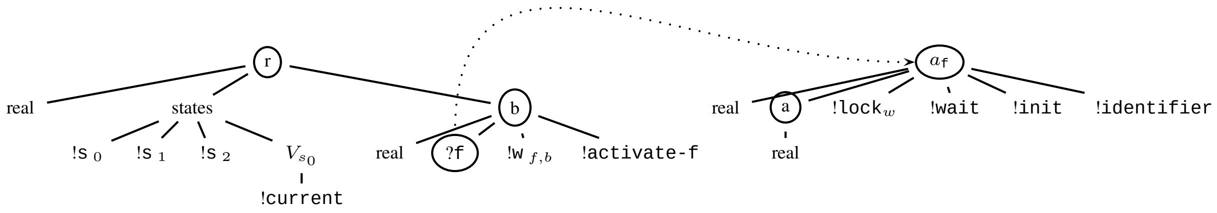
(1) Call and return of !init-valuation. This function returns the valuations for the variables associated with s_0. In this example, it returns the empty valuation, since s_0 has no associated variables.



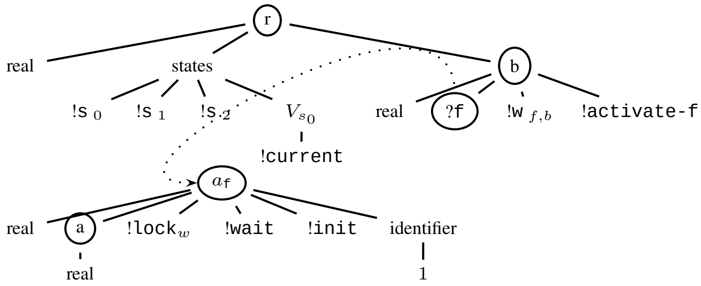
(2) Call and return of !lock$_f$. This function takes a lock on the system and checks if the activation of !f is allowed by the constraints of $S|A$. In our example, the activation is possible and the call to lock$_f$ returns !activate-f.



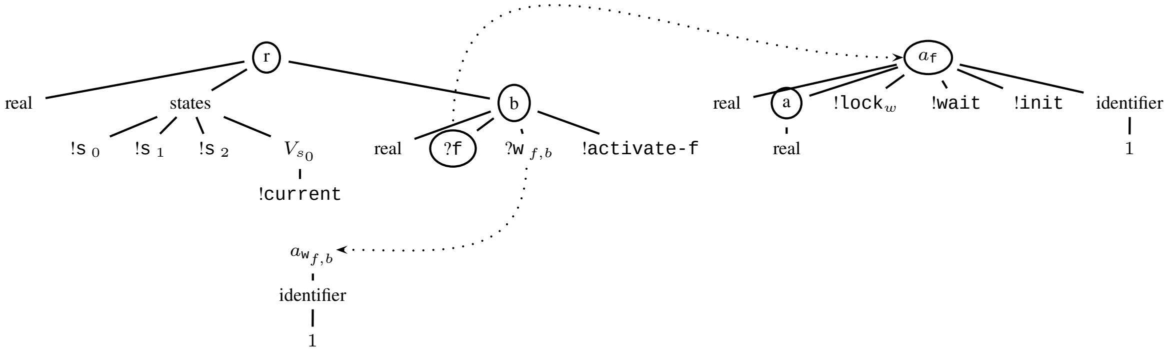
(3) Call of !f.



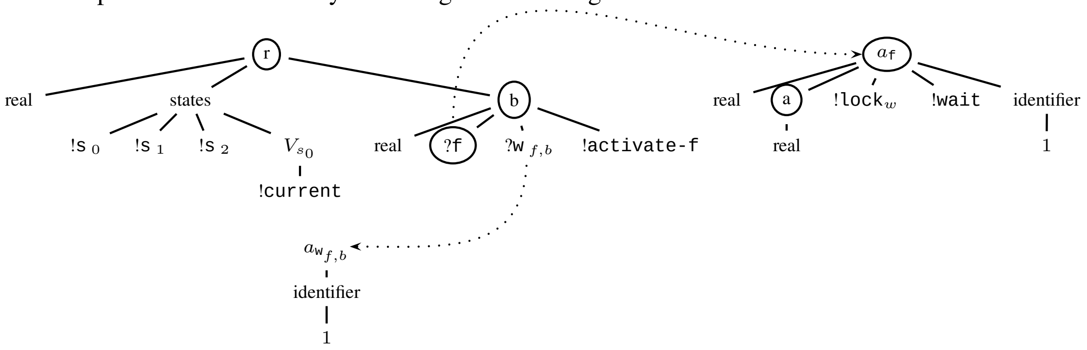
(4) The next few steps prepare the return of the call ?f. First, a fresh identifier for the workspace is created by a call to the external function identifier. For more clarity, we use an integer to represent the identifier.



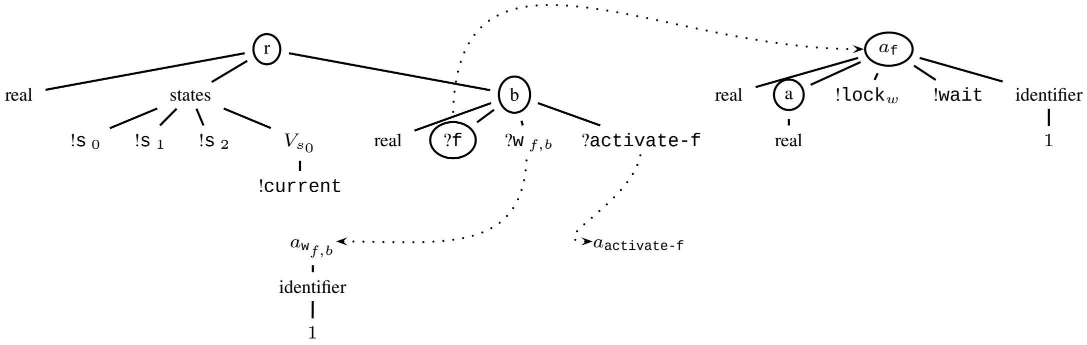
(5) A call of !w$_{f,b}$ copies the identifier of the workspace a$_{\mathtt{f}}$ using the fact that !init occurs in it.



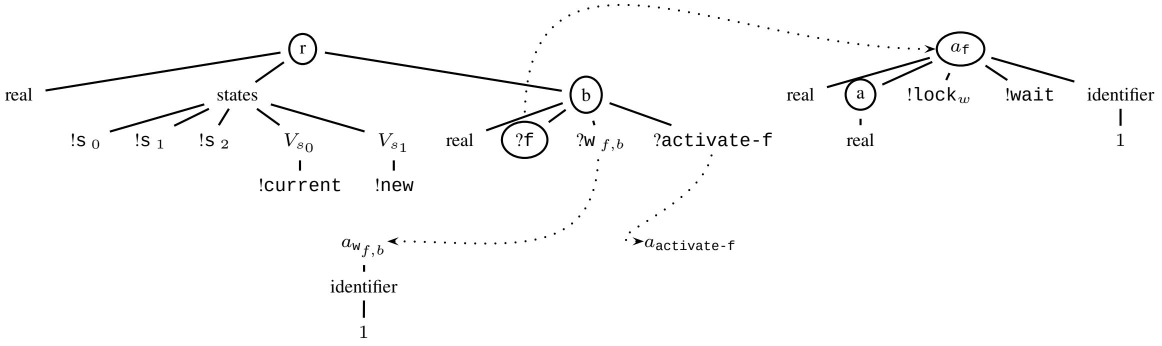
(6) The workspace of `f` is cleaned by activating and returning the function call `!init`.



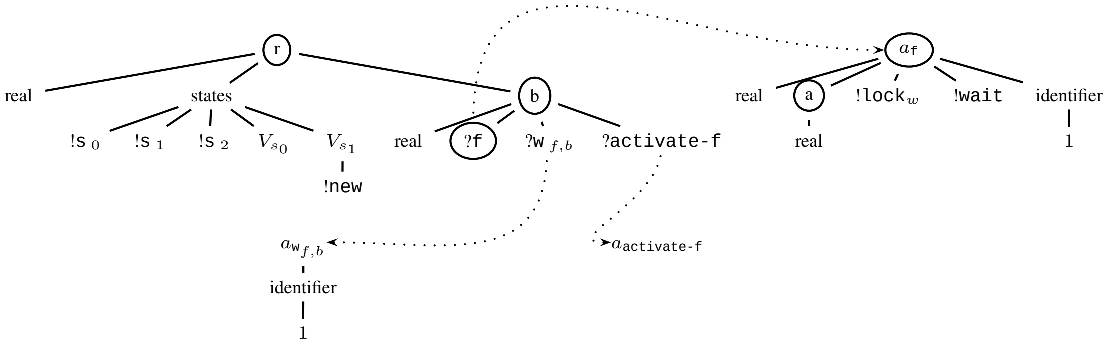
(7) Call of `activate-f`. This internal function can return only if there is no function call to `current` and there is some function call to `new`. This completes the simulation of the activation of `f`. It is followed by the simulation of the transition of the automaton, from s_0 to s_1.



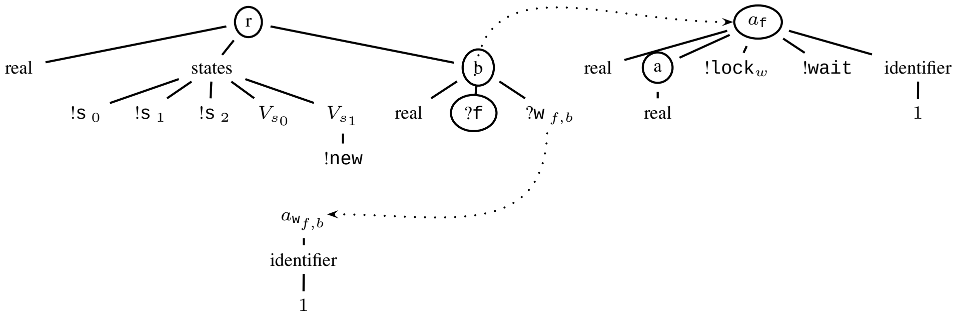
(8) In the general simulation, the function calls $!s_0$, $!s_1$, $!s_2$ are activated and returned to create new valuations for the associated variables. Since in our example there are no variables associated to states, the empty valuation corresponding to the state s_1 is represented by the subtree V_{s_1} with the function call `!new`.



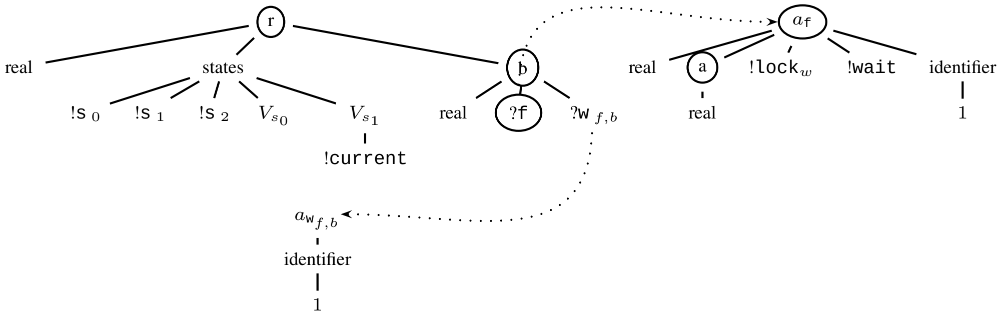
(9) The preceding valuations are marked as deleted. This is done by the call and return of the function current. In the example, the only such call occurs under the node labeled V_{s_0}.



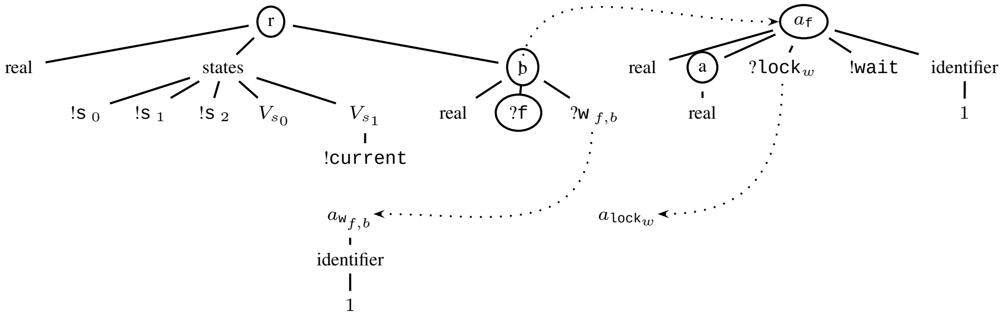
(10) ?activate-f can return only if there are no function calls to current but there is some function call to new.



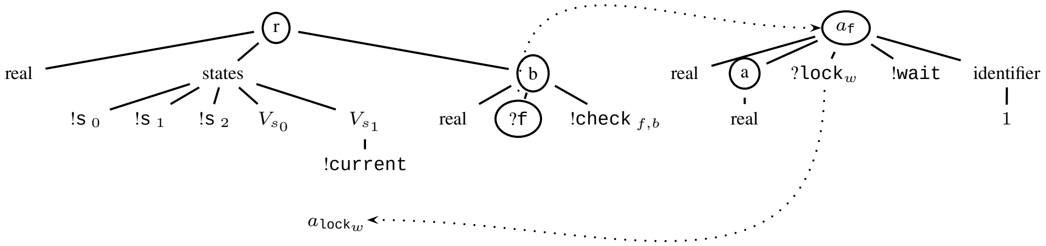
(11) The simulation of the first transition is completed by calling and returning the function call !new in order to obtain a function call !current.



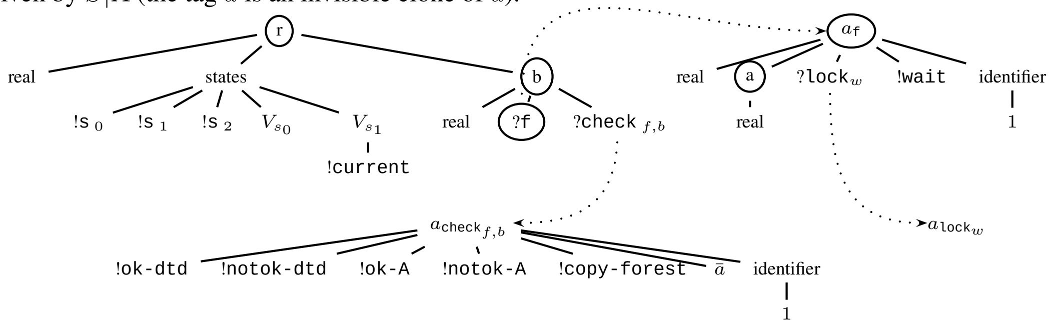
(12) The next steps simulate the return of ?f. The first step is the activation of !lock$_w$ in the workspace of f.



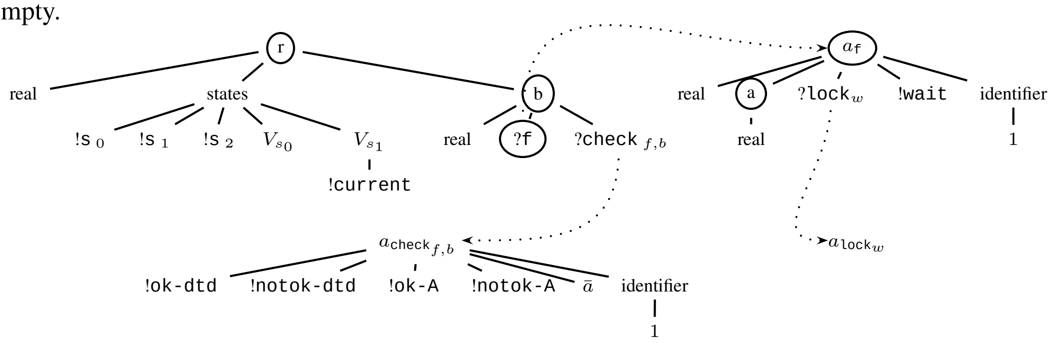
(13) The function call $\mathtt{w}_{f,b}$ associated with f returns by using the identifier of the workspace of f and the fact that the workspace of f has an activated function call ?lock$_w$.



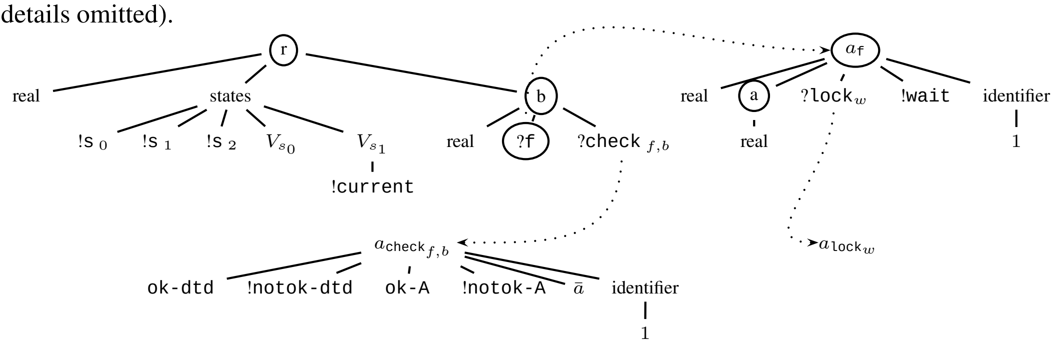
(14) The function call check$_{f,b}$ is activated to check that the return of f would be compatible with the constraints given by $S|A$ (the tag $\bar{a}$ is an invisible clone of a).



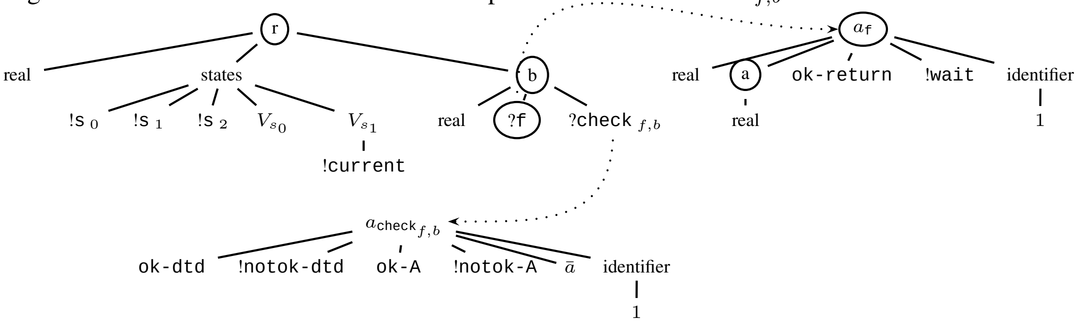
(15) First, the sibling subtrees of check$_{f,b}$ are copied in its workspace using the function copy-to under each visible tag and the function copy-forest in the workspace of check$_{f,b}$ (recall that the functions copy-to are omitted in the figures for readability, but are present under every visible node). In our example, the forest is empty.



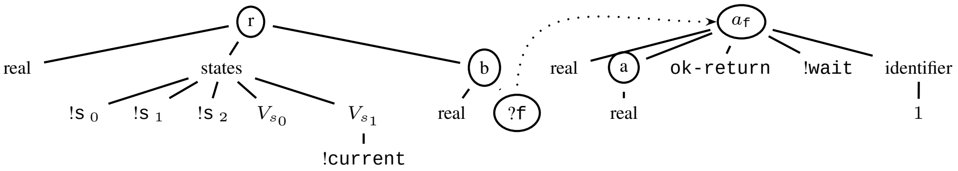
(16) Using the copied subtrees, the siblings of check$_{f,b}$ and the return of f built from the initial workspace, it is possible to check the correctness of the return of f for the DTD and the transition constraints. In our example, both are satisfied after the return. Then, !ok-dtd and ok-A are called and return ok-dtd and ok-A, respectively (details omitted).



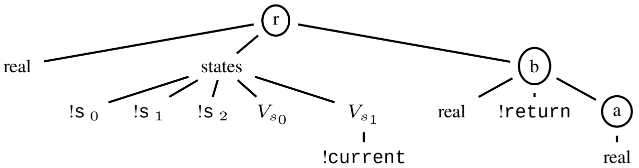
(17) The function call ?$\texttt{lock}_w$ returns the tag ok-return because of the presence of the labels ok-A and ok-DTD and using the common identifier found in the workspaces of ?f and ?$\texttt{check}_{f,b}$.



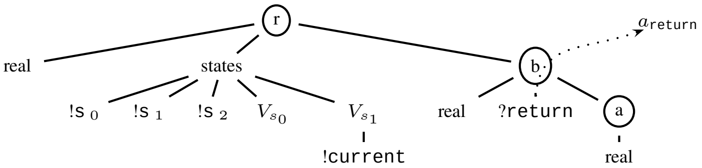
(18) The function call ?$\texttt{check}_{f,b}$ returns.



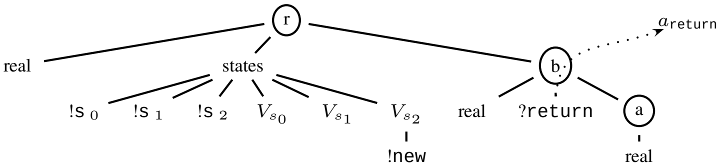
(19) The function call ?f returns because of the presence of the label ok-return in its workspace and the absence of ?$\texttt{check}_{f,b}$.



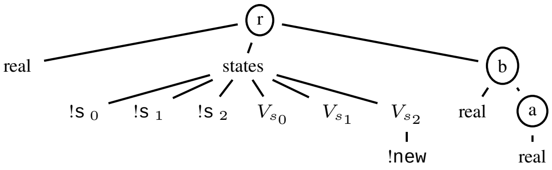
(20) The update of the state of the automaton begins as before. The function `return` plays the role of `activate-f` previously. It can return only if there is a call to `new` but no call to `current`.



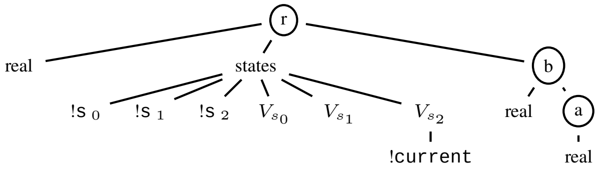
(21) The functions !$\texttt{s}_0$, !$\texttt{s}_1$, !$\texttt{s}_2$ are called and returned to obtain an empty valuation of s_2. The previous valuation is removed by calling and returning !`current`.



(22) The function call ?`return` returns.



(23) The update of the state is completed by calling `!new`, which returns `!current`.



This completes the simulation of the two transitions.

Simulation of TAXML^{sib} by BAXML

This follows from the simulation of AAXML^{sib} by GAXML and from Theorem 5.14, noting that the simulation of TAXML by AAXML does not introduce sibling calls to the same external function.

This concludes the proof of Theorem 5.8.

Remark 5.13. Theorem 5.8 shows that BAXML and GAXML can simulate AAXML^{sib} and TAXML^{sib} with respect to projection views. The converse is obviously false. Indeed, to see that BAXML (or GAXML) cannot be simulated by AAXML^{sib} or TAXML^{sib}, it is enough to consider a BAXML schema that produces an instance with two sibling calls to the same external function. By definition, such a schema cannot be simulated by AAXML^{sib} or TAXML^{sib}.

For AAXML and TAXML, we have the following.

THEOREM 5.14. *AAXML* $\hookrightarrow_{(id,\pi)}$ *TAXML and TAXML* $\hookrightarrow_{(id,\pi)}$ *AAXML.*

PROOF. We first show that AAXML $\hookrightarrow_{(id,\pi)}$ TAXML. Let $S|A$ be an AAXML schema with functions $\mathcal{F}_0$ and tags Σ_0. The broad lines of the simulation of AAXML by TAXML are similar to the simulation of AAXML^{sib} by GAXML. As in the latter case, the TAXML system must enforce an alternation of transitions and maintenance of the state/valuation information for A. This is done by a locking mechanism enforced by auxiliary functions, much like in the simulation by GAXML. We omit the similar details and focus on returns of external function calls.

Each function call notifies its return by a function call `!safe-r` that belongs to its answer (this can be enforced for external functions by their DTD). The function `!safe-r` works as a lock. To ensure that two sibling functions calls `?f` do not return consecutively, the TAXML formula imposes that no two consecutive instances contain a function call `!safe-r`. In particular, this requires the activation of `!safe-r`$_f$ in the instance following its first occurrence. The validity of the return with respect to A is checked, as in the simulation by GAXML, by the constraint $\varphi_{\textit{next}}$, whenever `!safe-r`$_f$ occurs (note that $\varphi_{\textit{next}}$ can be used directly in the Past-Tree-LTL formula).

We next show that TAXML $\hookrightarrow_{(id,\pi)}$ AAXML. To this end, we use a variant of AAXML, denoted by AAXML*. The automaton model of AAXML* differs from AAXML as follows:

(i) the automaton is equipped with final states, and a prerun must lead to some final state in order to be accepted,
(ii) the state variables are the same for all states and remain unchanged in each transition, and
(iii) the state variables range over the active domain of the entire prerun which is the input to the automaton, rather than just the last instance leading to that state.

We first show that TAXML can be simulated by AAXML*, then show how AAXML* can be simulated by AAXML.

From TAXML to AAXML* Let $\xi = \exists \bar{X} \varphi(\bar{X})$ be a Past-Tree-LTL formula. Recall that each such formula is obtained from a propositional Past-LTL formula $\bar{\varphi}$ with propositions P in which each proposition $p \in P$ is replaced by a Boolean combination of parameterized patterns ψ_p. Using a variant of the algorithm of [Vardi 1996] for finite words, one can construct a finite-state automaton $A_{\bar{\varphi}}$ whose alphabet consists of the truth assignments to P, that is equivalent to $\bar{\varphi}$. From this we can obtain an AAXML* automaton A_ξ equivalent to ξ as follows.

— For each truth assignment σ to P, let γ_σ be the Boolean combination of tree patterns obtained from the propositional formula $\wedge_{\sigma(p)=1} p \wedge_{\sigma(p)=0} \neg p$ by replacing each p by ψ_p
— For each state q of $A_{\bar{\varphi}}$, A_ξ has one state (q, σ) for each outgoing transition from q labeled σ, and transitions are induced by those in $A_{\bar{\varphi}}$. The state formula for (q, σ) is γ_σ. The state variables (which are all the same) equal $\bar{X}$.
— The final states of A_ξ are those of the form (q, σ) where q is final in $A_{\bar{\varphi}}$.

It is easily seen that the AAXML* automaton A_ξ is equivalent to ξ.

From AAXML* to AAXML We explain informally the main points in the simulation of AAXML* by AAXML. Consider an AAXML* specification $S^*|A^*$. We describe an AAXML specification $S|A$ that simulates it. Recall the differences (i)-(iii) between the AAXML* and AAXML automata. The simulation by $S|A$ is similar to the maintenance of the set of reachable states and valuations, used in the simulation of AAXML by GAXML. Dealing with (i) and (ii) is straightforward. To account for the final states, $S|A$ must check that at each transition, one of the reachable states is final. The fact that state variables are the same and do not change in A^* is easily enforced in A using equalities among variables of consecutive states. The most delicate part of the simulation concerns (iii), i.e. the difference in the active domain semantics for the two models. Indeed, at any given transition in the prerun, state formulas are evaluated on the active domain of the entire prerun. This includes values occurring in *past* instances and values occurring in *future* instances (introduced by external functions). We discuss both in turn.

Dealing with past values is fairly straighforward. It is sufficient to ensure that at any point, the current active domain contains all values of previous instances in the prerun. To this end, we use a new internal, continuous function `collect`, whose role is to maintain the cumulative active domain of the instances in the prerun. More precisely, the DTD of S^* is modified so that !`collect` or ?`collect` must occur under a node labeled *values* (a new tag) which in turn occurs under the root. The argument query of `collect` produces all data values in the current instance, and the answer query returns all data values in its workspace. The pattern automaton A^* is modified as follows. For each state p that has at least one outgoing edge, we introduce two new intermediate states, p_1 and p_2, with the same number of associated variables as p. The role of p_1 and p_2 is to force an activation of !`collect`, followed by a return of ?`collect`, before any other transition. The state formula of p_1 is $\Upsilon'(p)(\overline{X}_p/\overline{X}_{p_1}) \wedge \alpha_1$, where $\Upsilon'(p)(\overline{X}_p/\overline{X}_{p_1})$ is obtained from $\Upsilon(p)$ by modfiying each pattern in order to force all matchings to avoid the subtree rooted at *values*, and by replacing the variables $\overline{X}_p$ with $\overline{X}_{p_1}$, and α_1 checks the existence of ?`collect`. There is an edge from p to p_1 and $\delta(p, p_1)$ makes all variables $\overline{X}_{p_1}$ equal to $\overline{X_p}$. Similarly, the state formula of p_2 is $\Upsilon'(p)(\overline{X}_p/\overline{X}_{p_2}) \wedge \alpha_2$, where α_2 checks the existence of !`collect` (which means that the call to `collect` has returned) and $\delta(p_1, p_2)$ makes all variables $\overline{X}_{p_1}$ equal to $\overline{X}_{p_2}$. Finally, for each state q of A^* such that $\delta(p, q)$ is defined, $\delta(p_2, q)$ is obtained from $\delta(p, q)$ by replacing $\overline{X}_p$ with $\overline{X}_{p_2}$. It is clear that the intermediate states ensure that the cumulative active domain of the prerun up to the current instance is found under the node labeled *values* after each visible transition is simulated.

It now remains to deal with new values introduced in *future* instances of the prerun, relative to the current instance. These may arise from answers to external function calls. We make use of the previous construction ensuring that the cumulative active domain of the prerun up to the current instance is maintained under the distinguished node labeled *values*. Handling future values is trickier, because the semantics requires taking these into account in previous transitions. Dealing with this requires augmenting the state/valuation maintenance algorithm. Specifically, $S|A$ must decide if the current transition would be allowed had A^* been run from the beginning on the active domain extended with the new values. In order to do this incrementally (without re-running the automaton on the extended domain), A must maintain some additional information summarizing the reachable states and valuations, where the latter include values outside the current prerun. In order to do this, the key observation is that a positive pattern with a free variable X cannot be satisfied for any value of X not in the current instance. Let @ be a new data constant, representing an arbitrary value outside the current active domain. Consider a valuation ν of the state variables $\overline{X}$ into the cumulative active domain augmented with @. Let us call a valuation *indefinite* if it maps at least one variable to @, and *definite* otherwise. We can define the satisfaction of a tree pattern $P(\nu(\overline{Y}))$ in a BAXML instance, where $\overline{Y} \subseteq \overline{X}$, as follows: if $\nu(\overline{Y})$ is definite, then satisfaction is defined as usual; otherwise, $P(\nu(\overline{Y}))$ is not satisfied. This extends to satisfaction of Boolean combinations of tree patterns, so of state formulas. The maintenance algorithm is now be extended to keep states together with definite and indefinite valuations. When a transition from instance I and state p to instance J and state q is simulated, the following is done:

(i) the set of definite valuations for p is augmented by adding, for each indefinite valuation ν of $\overline{X}_p$, all valuations $\nu \circ \nu'$, where ν' maps @ to any value in the active domain of J that is not in the cumulative active domain up to I;
(ii) the maintenance algorithm computes in the usual way the set of possible definite valuations for q, using the set of definite valuations computed in (i) for p;
(iii) a new set of indefinite valuations is computed for q, using J and $\Upsilon(q)(\overline{X}_q)$.

Let $S|A$ be the AXML schema implementing the extended maintenance algorithm. It is clear that $S|A$ simulates $S^*|A^*$. □

From the proof of Theorem 5.14 we have the following.

COROLLARY 5.15. *TAXML^{sib} $\hookrightarrow_{(id,\pi)}$ AAXML^{sib} and TAXML^{sib} $\hookrightarrow_{(id,\pi)}$ AAXML^{sib}.*

PROOF. It can be checked that the simulations described in the proof of Theorem 5.14 preserve the sibling restriction on external functions. □

The proofs of the above results provide insight into the simulations of the various languages, and in particular highlight 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 15.

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)

Fig. 15: Cost of various simulations in Theorems 5.8 and 5.14

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

PROOF. We first show that there exists an AAXML schema with external functions that cannot be simulated by a GAXML schema relative to a projection view. Intuitively, if there are several sibling active function calls to the same external function, the GAXML schema is not able to impose that only one function call returns before the states of the automaton are updated and validity of the transition is ensured.

The AAXML schema $S|A$ is the following. We describe the shape of a run. The initial instance is a tree rooted at r with one child labeled by a continuous function !g. The function !g returns an external, non-continuous function call !f. Repeated calls to g and f (in alternation) generate an unbounded number of sibling calls ?f. Each function f returns a label a. The automaton further imposes that no more than one answer to ?f be returned in a run.

We show that there is no GAXML schema simulating $S|A$. Assume towards a contradiction that there exists such a schema $S'|\gamma$. Let M be the maximum integer used in the DTD of S'. We exhibit a prerun that is valid for $S'|\gamma$, but whose projection is not valid for $S|A$. First, let $\rho = (I_0, e_0) \cdots (I_m, e_m)$ be a prerun for $S|A$ in which I_m has $M + 1$ occurrences of ?f and e_m is the only return of a call ?f occurring in ρ. Let I be the instance resulting from the return of another call ?f of I_m (let e be this event). Note that ρ is a valid prerun of $S|A$ whereas $\rho.(I, e)$ is not. Nonetheless, we show that $\rho.(I, e)$ is the projection of a prerun of $S'|\gamma$. Since $S'|\gamma$ simulates $S|A$ and ρ is a prerun of $S|A$, there exists a prerun of $S'|\gamma$ with a subsequence $(I'_{i_0}, e'_{i_0}) \cdots (I'_{i_m}, e'_{i_m})$ so that $i_0 = 0, i_m = m$ and (I_j, e_j) is the projection of (I'_{i_j}, e'_{i_j}), $0 \leq j \leq m$. In particular, $I'_{i_{m-1}}$ contains $M + 2$ calls to ?f, I'_{i_m} contains $M + 1$ calls to ?f, and (since calls ?f are visible), I'_{i_m} is obtained from $I'_{i_m - 1}$ by the return of a call to ?f, consisting of some forest F. We claim that $S'|\gamma$ allows the transition from I'_{i_m} to I' in which another call to ?f returns the same forest F. Indeed, because in the BAXML semantics isomorphic subtrees are reduced, the two occurrences of F are merged so the only difference between I_{i_m} and I' is that I_{i_m} has $M+1$ calls ?f whereas I' has M such calls. Since M is the maximum integer used in the DTD of S', and I_{i_m} satisfies the DTD, so does I'. Similarly, I_{i_m} and I' satisfy the same tree patterns because the two instances are homomorphic to each other. Thus, I' satisfies all static constraints of S'. Since external function returns have no guards, the transition is valid in $S'|\gamma$. However, the projection of I' is I and, as we have seen, $\rho.(I, e)$ is not a valid prerun of $S|A$. This contradicts the existence of $S'|\gamma$.

The fact that TAXML cannot be simulated by GAXML follows from the fact that AAXML can be simulated by TAXML (Theorem 5.14) and AAXML cannot be simulated by GAXML. The difficulty is the same as in the above proof. □

$$\begin{tikzcd}
\text{TAXML} \arrow[r, leftrightarrow] & \text{AAXML} \\
\text{BAXML} \arrow[u] \arrow[r, leftrightarrow] \arrow[ur] & \text{GAXML} \arrow[u] \arrow[ul] \\
\text{TAXML}^{sib} \arrow[u] \arrow[r, leftrightarrow] \arrow[ur] & \text{AAXML}^{sib} \arrow[u] \arrow[ul]
\end{tikzcd}$$

Fig. 16: Summary of the simulation results

The simulation results of this section relative to projection views are summarized in Figure 16 (single arrows indicate simulation only in one direction, and double arrows indicate mutual simulation).

Comparison with coarser views

We have focused in this section on simulation relative to projection views (id, π). The results obtained turn out to be quite powerful. Indeed, by Lemma 2.5, the positive results extend to any views that are coarser than projection 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 2.5, the positive simulation results shown in Figure 16 continue to hold relative to $(id \circ V, \pi \circ V)$.

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 same simulation results hold.

6. BAXML AND TUPLE ARTIFACTS

In the previous section, we compare 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 [Deutsch et al. 2009]. 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 they 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, 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 [Deutsch et al. 2009] for a detailed example of an artifact system.

The Tuple Artifact Model

We provide the definition of the tuple artifact model, adapted from [Deutsch et al. 2009]. A relational database schema $\mathcal{D}$ consists of a finite set of relation symbols with specified arities. The arity of relation R is denoted by *arity*(R). An instance, or interpretation, over a database schema, is a mapping associating to each relation symbol R of the schema a finite relation over D, of arity *arity*(R). We assume familiarity with First-Order logic (FO) over database schemas. Given a schema $\mathcal{D}$, $\mathcal{L}_{\mathcal{D}}$ denotes the set of FO formulas over $\mathcal{D}$. If $\varphi(\bar{x})$ is an FO formula with free variables $\bar{x}$, and $\bar{u}$ is a tuple over D of the same arity as $\bar{x}$, we denote by $\varphi(\bar{u})$ the sentence obtained by substituting $\bar{u}$ for $\bar{x}$ in $\varphi(\bar{x})$. Note that, since D is infinite, an FO formula $\varphi(\bar{x})$ may be satisfied by infinitely many tuples $\bar{u}$ over D (so may define an infinite relation). Finiteness and effective evaluation can be guaranteed by using the *active domain semantics*, in which the domain is restricted to the set of elements occurring in the given instance (sometimes augmented with a specified finite set of constants in D, which by default is empty). For an instance I, we denote its active domain by *adom*(I). Unless otherwise specified, we assume active domain semantics for quantified variables and unrestricted semantics for the free variables of a formula.

The artifact model uses a specific notion of class, schema and instance, defined next.

Definition 6.1. An *artifact class* is a pair $\mathcal{C} = \langle R, S\rangle$ where R and S are two relation symbols. An *instance* of $\mathcal{C}$ is a pair $C = \langle \mathsf{R}, \mathsf{S}\rangle$, where *(i)* R, called *attribute relation*, is an interpretation of R containing exactly one tuple over D, and *(ii)* S, called *state relation*, is a finite interpretation of S over D.

We also refer to an *artifact instance of class* $\mathcal{C}$ as *artifact instance*, or simply *artifact* when the class is clear from the context or irrelevant.

Definition 6.2. An *artifact schema* is a tuple

$$\mathbb{A} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{DB}\rangle$$

where each $\mathcal{C}_i = \langle R_i, S_i\rangle$ is an artifact class, $\mathcal{DB}$ is a relational schema, and for all $i \neq j$, $\mathcal{C}_i$, $\mathcal{C}_j$, and $\mathcal{DB}$ have no relation symbols in common.

By slight abuse, we sometimes identify an artifact schema $\mathbb{A}$ as above with the relational schema

$$\mathcal{DB}_{\mathbb{A}} = \mathcal{DB} \cup \{R_i, S_i \mid 1 \leq i \leq n\}.$$

An instance of an artifact schema is a tuple of class instances, each corresponding to an artifact class, plus a database instance:

Definition 6.3. An *instance* of an artifact schema

$$\mathbb{A} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n, \mathcal{DB}\rangle$$

is a tuple $A = \langle C_1, \ldots, C_n, DB\rangle$, where C_i is an instance of $\mathcal{C}_i$ and DB is an instance of $\mathcal{DB}$ over D.

Again by slight abuse, we identify each instance

$$A = \langle C_1, \ldots, C_n, DB\rangle$$

of $\mathbb{A}$ with the relational instance $DB \cup \{\mathsf{R_i}, \mathsf{S_i} | 1 \leq i \leq n\}$ over schema $\mathcal{DB}_{\mathbb{A}}$. Let $\mathbb{A}$ be an artifact schema and $\mathcal{DB}_{\mathbb{A}}$ its relational schema. Given an artifact instance over $\mathbb{A}$, the semantics of formulas in $\mathcal{L}_{\mathbb{A}}$ is the standard semantics on the associated relational instance over $\mathcal{DB}_{\mathbb{A}}$.

We now define the syntax of services. It will be useful to associate to each attribute relation R of an artifact schema $\mathbb{A}$ a fixed sequence $\bar{x}_R$ of distinct variables of length *arity*(R).

Definition 6.4. A *service* σ over an artifact schema $\mathbb{A}$ is a tuple $\sigma = \langle \pi, \psi, \mathbf{S}\rangle$ where:

— π, called *pre-condition*, is a sentence in $\mathcal{L}_{\mathbb{A}}$;
— ψ, called *post-condition*, is a formula in $\mathcal{L}_{\mathbb{A}}$, with free variables
  $\{\bar{x}_R \mid R$ is an attribute relation of an artifact class in $\mathbb{A}\}$;
— **S** is a set of *state rules* of the form:
  — $S(\bar{x}) \leftarrow \varphi_S^+(\bar{x})$;
  — $\neg S(\bar{x}) \leftarrow \varphi_S^-(\bar{x})$;
  where S is a state relation of $\mathbb{A}$, $\varphi_S^+(\bar{x})$ and $\varphi_S^-(\bar{x})$ are $\mathcal{L}_{\mathbb{A}}$-formulas with free variables $\bar{x}$ s.t. $|\bar{x}| =$ *arity*(S).

As seen below, the formulas $\varphi_S^+(\bar{x})$ and $\varphi_S^-(\bar{x})$ are used to define updates to the state relation S when the service is applied. The formula $\varphi_S^+(\bar{x})$ defines the tuples to be inserted, and $\varphi_S^-(\bar{x})$ the tuples to be deleted (see below). If a formula is not provided for a state relation S, the set of tuples to be inserted or deleted is taken to be empty.

Definition 6.5. An *artifact system* is a pair $\Gamma = \langle \mathbb{A}, \Sigma \rangle$, where $\mathbb{A}$ is an artifact schema and Σ is a non-empty set of services over $\mathbb{A}$.

We next define the semantics of services. We begin with the notion of possible successor of a given artifact instance with respect to a service.

Definition 6.6. Let $\sigma = \langle \pi, \psi, \mathbf{S} \rangle$ be a service over artifact schema $\mathbb{A}$. Let A and A' be instances of $\mathbb{A}$. We say that A' is a *possible successor* of A with respect to σ (denoted by $A \xrightarrow{\sigma} A'$) if the following hold:

(1) $A \models \pi$;
(2) $A'|\mathcal{DB} = A|\mathcal{DB}$ (A and A' agree on all relations in $\mathcal{DB}$);
(3) $A, \nu \models \psi$, where ν is the valuation of the free variables of ψ mapping $\bar{x}_R$ to $\bar{u}_R$ for each attribute relation R of $\mathbb{A}$;
(4) for each state relation S of $\mathbb{A}$ and tuple $\bar{u}$ over $adom(A)$ of arity *arity*(S), $A' \models S(\bar{u})$ iff

$$\begin{array}{rl} A \models & (\varphi^+_S(\bar{u}) \wedge \neg\varphi^-_S(\bar{u})) \vee (S(\bar{u}) \wedge \varphi^+_S(\bar{u}) \wedge \varphi^-_S(\bar{u})) \\ & \vee (S(\bar{u}) \wedge \neg\varphi^+_S(\bar{u}) \wedge \neg\varphi^-_S(\bar{u})) \end{array}$$

where $\varphi^+_S(\bar{u})$ and $\varphi^-_S(\bar{u})$ are interpreted under active domain semantics, and are taken to be false if the respective rule is not provided. Thus, the new state relation S is obtained by inserting the tuples defined by φ^+_S and deleting those defined by φ^-_S, with deletion given priority over insertion in case of conflict, except for tuples previously in S, which are preserved in case of conflict.

Note that, according to (2) in Definition 6.6, services do not update the database contents (thus, the database contents is fixed throughout each run, although it may of course be different across runs). Instead, the data that is updatable throughout a run is carried by the artifacts themselves, as attribute and state relations. Note that, if desired, one can make the entire database updatable by turning it into a state. Also observe that the distinction between state and database is only conceptual, and does not preclude implementing all relations within the same DBMS.

We next define the notion of run of an artifact system $\Gamma = \langle \mathbb{A}, \Sigma \rangle$. An *initial instance* of Γ is an artifact instance over $\mathbb{A}$ whose states are empty.

Definition 6.7. A *prerun* of an artifact system $\Gamma = \langle \mathbb{A}, \Sigma \rangle$ is a finite sequence $\rho = \{(\rho_i, \sigma_i)\}_{0 \leq i \leq n}$ where each ρ_i is an artifact instance over $\mathbb{A}$ and each σ_i is a service, such that:

— ρ_0 is an initial instance of Γ;
— for each $i > 0$, $\rho_{i-1} \xrightarrow{\sigma_i} \rho_i$.

We say that a pre-run is *blocking* if its last configuration has no possible successor. As for BAXML, blocking runs are extended by repeating forever the last configuration, with corresponding transitions labeled *block*. A *run* is an infinite sequence $\{(\rho_i, \sigma_i)\}_{i \geq 0}$ in which either every finite prefix is a prerun, or the run is obtained by extending a blocking prerun by repeating forever the last configuration with transitions labeled *block*. For an artifact system, the associated *workflow system* is defined from the set of runs analogously to BAXML. In particular, the states are artifact instances, and the events are services causing state transitions or the special event *block*.

Workflow system semantics. The workflow system semantics of artifact systems is defined from its runs analogously to the semantics of BAXML, GAXML, AAXML, and TAXML (Section 5). For each artifact system Γ, the nodes of its associated workflow system are the finite prefixes of runs of Γ. The root is the empty prefix, and its state label is the empty instance. The state label for each node other than the root is the last instance in the prefix. For each non-root node ν, there is an edge labeled σ from ν to node ν' if ν' extends ν with a single instance obtained by application of the service σ. The root has an outgoing edge to each node consisting of a prefix of length one, labeled by a distinguished event *init*. Thus, transitions from the root provide the initial instances of runs, and the infinite paths starting from children of the root correspond to the runs of Γ. Because of the semantics of blocking runs, each path is extensible to an infinite path.

Simulation of Tuple Artifacts by BAXML

We denote the tuple artifact model by $\mathcal{TA}$. More precisely, $\mathcal{TA}$ is the set of all artifact system specifications, with workflow system semantics.

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 instances. . A relation R with attributes $A_1, \ldots, A_m$ is naturally represented in BAXML by a subtree rooted at R, satisfying the

DTD below, denoted by Δ_R:

$$\begin{array}{lll} R & \rightarrow & |\mathit{tup}_R| \geq 0 \\ \mathit{tup}_R & \rightarrow & \wedge_{i=1}^{m} |A_i| = 1 \\ A_i & \rightarrow & |\mathit{dom}| = 1 \end{array}$$

Given an artifact instance, we refer to the contents of the artifact relations, consisting of single tuples, as the *artifact tuples*. 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.

We define the class of views used in the simulation, denoted by $\mathcal{V}_{\mathcal{TA}}$. Each view is defined relative to a set $\mathcal{R}$ of tags and a set F of function names. Intuitively, the tags in $\mathcal{R}$ are meant to label subtrees encoding relations, as above. We say that a BAXML workflow system is $\mathcal{R}$-relational if for each $R \in \mathcal{R}$ there exists a DTD Δ_R of the above shape such that each BAXML instance labeling a non-root state of the workflow system contains exactly one occurrence of each tag R in $\mathcal{R}$, and the subtree rooted at R satisfies Δ_R. The view $V_{\mathcal{R},F}$ in $\mathcal{V}_{\mathcal{TA}}$ is defined as follows. If the workflow system is *not* $\mathcal{R}$-relational, then all state labels are mapped to $\emptyset$ and all edge labels are mapped to ϵ (these workflow systems are irrelevant because they are not used in the simulation). If the workflow system is $\mathcal{R}$-relational, the view is defined as follows:

— BAXML instances labeling non-root states are mapped to the relational instance represented by the subtrees rooted at labels in $\mathcal{R}$;
— events consisting of calls to functions in F are mapped to the name of the function;
— the *init* event is preserved; and,
— all other events are mapped to ϵ.

The main result is the following.

THEOREM 6.8. $\mathcal{TA} \hookrightarrow_{(\mathrm{id},\mathcal{V}_{\mathcal{TA}})}$ *BAXML. In other words, for each* $\mathcal{TA}$ *system* Γ *there exists a BAXML schema* S *and a view* $V \in \mathcal{V}_{\mathcal{TA}}$ *such that* $[\Gamma] \sim V([S])$.

PROOF. We show that $\mathcal{TA}$ can be simulated by GAXML. This suffices, because BAXML can simulate GAXML. In more detail, suppose that $\mathcal{TA} \hookrightarrow_{(id,\mathcal{V}_{\mathcal{TA}})}$ GAXML. By Theorem 5.8, GAXML $\hookrightarrow_{(id,\pi)}$ BAXML. By Lemma 2.6, (since $\mathcal{V}_{\mathcal{TA}} = id \circ \mathcal{V}_{\mathcal{TA}}$) $\mathcal{TA} \hookrightarrow_{(id,\pi\circ\mathcal{V}_{\mathcal{TA}})}$ BAXML. From the definitions of π and $\mathcal{V}_{\mathcal{TA}}$, it is clear that $\pi \circ \mathcal{V}_{\mathcal{TA}} \subseteq \mathcal{V}_{\mathcal{TA}}$. Thus, $\mathcal{TA} \hookrightarrow_{(id,\mathcal{V}_{\mathcal{TA}})}$ BAXML.

We sketch the simulation of $\mathcal{TA}$ by GAXML for artifact systems with only one artifact class with a single state and database relation, and a single service. This is sufficient to capture the salient elements of the simulation. As discussed in [Deutsch et al. 2009], an arbitrary $\mathcal{TA}$ system can be easily represented by such a restricted system.

Suppose the artifact system has an artifact tuple with k attributes $A_1, \ldots, A_k$, a database relation *DB*, and a state relation S. The unique service has pre-condition π, postcondition ψ, and state formulas φ_S^+ and φ_S^-. Relations will be represented in the simulating GAXML system in the standard way, by subtrees of bounded depth (see Section 6). The database relation is a fixed subtree in the main document, while the state and artifact tuple are represented in workspaces of function calls, which facilitates updating their values. More specifically, the state is represented and updated using the workspaces of two function calls that alternate between carrying the current state and computing the next state.

An application of the service requires simulating the following:

(1) evaluating the pre-condition π on the database, current state and current artifact tuple.
(2) evaluating the FO formulas φ_S^+ and φ_S^- and generating the new S in the workspace of one of the two functions mentioned above.
(3) non-deterministically generating a new candidate artifact tuple and verifying satisfaction of the postcondition ψ.

The bookkeeping needed to enforce the above sequencing can be straightforwardly done with auxiliary functions. There are two delicate points: the evaluation of an FO formula, and simulating (3) so that all qualified next artifact tuples can be generated and failed attempts do not lead to spurious blocking or infinite chains of ϵ-transitions. Recall that in general there are infinitely many new candidate artifact tuples, because new values can come from the infinite domain D.

Evaluating an FO formula. We first elaborate on the evaluation of FO formulas. Recall that the formulas φ_S^+, φ_S^-, and π are interpreted with active domain semantics. Consider an FO formula written using $\wedge, \neg, \exists$. The formula is evaluated by structural recursion on its syntax tree. Given standard representations of the result of two subformulas, it is easy to compute the relation obtained by applying $\wedge$ and $\exists$. Applying $\neg$ is trickier. For conciseness, we illustrate how to compute the complement of a unary relation P with respect to the active domain (this can be easily extended

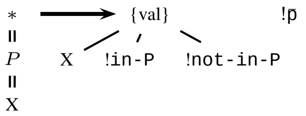


Fig. 17: The query of *check*$_P$

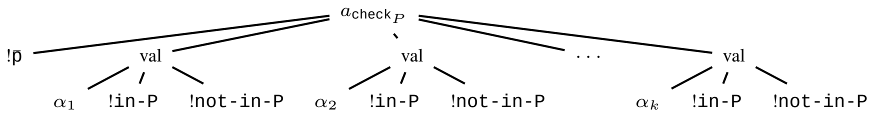


Fig. 18: Shape of the initial workspace of *check*$_P$

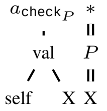


Fig. 19: Guard of the function in-P

to arbitrary arity). The relation P is represented by a subtree with root labeled P, satisfying the DTD

$$P \rightarrow |dom| \geq 0.$$

The complement is constructed as follows. First, a call to a function !check$_P$ generates the current active domain, where each value is adorned with two functions !in-P and !not-in-P. More precisely, the argument query of !check$_P$ is shown in Figure 17 and its initial workspace is of the form shown in Figure 18. In this example, a data value is denoted by α_i (the role of !p̄ will be explained shortly).

The functions in-P and not-in-P are internal. The call guard of in-P, shown in Figure 19, verifies that the value adjacent to the call is in P, whereas the guard of not-in-P checks that the value is *not* in P (so the guard of not-in-P is the negation of the guard of in-P). The functions in-P and not-in-P return, respectively, a label in and a label *not-in*. The role of the function !p̄ is dual. First, its guard ensures that for each value, one of its siblings !in-P or !not-in-P has been called and has been returned. To this end, its guard forbids the presence of two siblings !/?in-P and !/?not-in-P. Second, its argument query computes the complement of P, by collecting the values with a sibling *not-in*. The argument query of p̄ is shown in Figure 20.

Generating the new artifact tuple. Like the state, the artifact tuple is represented and updated using the workspaces of two functions that alternate between carrying the current value and computing the new value of the artifact tuple. Recall that generally there are infinitely many candidates for the next artifact tuple, since the free variables of the post-condition range over the infinite domain D. Observe that satisfaction of the post-condition is invariant under the following equivalence relation on k-tuples over D: $\langle a_1, \ldots, a_k\rangle \equiv \langle b_1, \ldots, b_k\rangle$ iff for all i, j:

— $a_i = a_j$ iff $b_i = b_j$,
— if either a_i or b_i is in the active domain, then $a_i = b_i$.

To each equivalence class corresponds a *type* specifying the values for the coordinates that belong to the active domain, and the equality type for the coordinates whose values are not in the active domain. It is straightforward to nondeterministically construct a relation containing one representative tuple for each equivalence class. Specifically, internal function calls are used to generate the values of the coordinates in the active domain, and external functions to generate values for the coordinates outside the active domain. The equality type for the latter is imposed by constraints. In addition, each tuple is adorned with a function call whose role is to evaluate the post-condition ψ for the tuple, returning *ok* in the affirmative and *not-ok* in the negative. Since ψ is in FO, this can be done similarly to the above. The functions evaluating ψ for each tuple are called non-deterministically, and a simple locking mechanism ensures that (i) the functions are evaluated completely one at a time, and (ii) function activations are blocked in the current round as soon as one of them returns *ok*. The new artifact tuple is the unique one marked *ok*. It can be easily checked that every candidate tuple can be generated in this manner by some computation path. If there is no such tuple, the artifact system blocks, and so does the simulation. □

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,

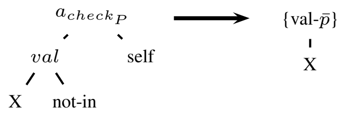


Fig. 20: Argument query of function $\bar{\mathsf{p}}$

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

PROOF. The proof is based on a reduction from the implication problem for functional and inclusion dependencies (FDs and IDs), known to be undecidable. Specifically, we consider instances of the implication problem of the form $\Delta \models f$, where Δ is a set of FDs and IDs, and f an FD. We consider a BAXML schema S whose initial instance consists of a single external function !e under the root. The function returns a tree representing an arbitrary finite relation, of the form shown in Figure 6. Specifically, each tuple is adorned with one function !f_τ for each ID τ in Δ. Additionally, there is one function !g under the root R. The call guard of each f_τ checks that the ID τ is *violated* for the sibling tuple. Satisfaction of the FDs in Δ, and violation of f, are ensured by static constraints. The guard of !g simply checks that the relation returned by the call to !e is non-empty.

We consider the view V_S retaining all functions. It is easy to check that $\Delta \not\models f$ iff there is a blocking run of S whose view under V_S is $\rho = init.\mathsf{e}.\mathsf{g}.(\mathit{block})^\omega$ (we ignore the constant state). Indeed, since no function !f_τ can be called, all IDs in Δ are satisfied. Recall that satisfaction of the FDs in Δ and violation of f are ensured by the constraints. Thus, the non-empty instance returned by e satisfies Δ and violates f.

Now suppose towards a contradiction that one can effectively construct, for each BAXML schema as above, a corresponding artifact system Γ with a view $V_\Gamma \in \mathcal{V}_{serv}$ so that $V_S([S]) \sim V_\Gamma([\Gamma])$. By definition, the first event in both $[S]$ and $[\Gamma]$ is init. Also, in $[S]$ there is a unique edge labeled init, leading to the node whose state is *root*/!e. Let $T_{!e}$ be the subtree of $[S]$ rooted at that node. By definition of $\sim$, $V_S(T_{!e})$ must be w-bisimilar to $V(T)$ for every subtree $T \in \mathcal{T}_{init}$, where $\mathcal{T}_{init}$ consists of the subtrees of $[\Gamma]$ whose roots have incoming edge init. In other words, Γ must simulate S regardless of its database. In particular, this must be the case for the empty database. Thus, let $T_\emptyset$ be the subtree in $\mathcal{T}_{init}$ corresponding to the empty database. From the above it follows that $V_S(T_{!e}) \sim V_\Gamma(T_\emptyset)$.

Recall that $\Delta \not\models f$ iff $V_S(T_{!e})$ contains a path from the root labeled $\mathsf{e}.\mathsf{g}.\mathit{block}$. Since $V_S(T_{!e}) \sim V_\Gamma(T_\emptyset)$, this happens iff $V_\Gamma(T_\emptyset)$ contains a path from the root labeled $\epsilon^*.\mathsf{e}.\epsilon^*.\mathsf{g}.\epsilon^*.\mathit{block}$. By definition of $\sim$, since $V_S(T_{!e})$ has no infinite branches of ϵ-transitions (in fact no ϵ-transitions at all), $V_\Gamma(T_\emptyset)$ may not have infinite branches of ϵ-transitions. Also note that $T_\emptyset$ is finitely branching, modulo isomorphism (this is because in artifact systems, each transition other than init generates only finitely many non-isomorphic states from each given state). It follows that from each given node, the set of lengths of ϵ-paths originating at that node is bounded (otherwise, an easy induction shows that there must be an infinite path of ϵ-transitions from that node). This allows to effectively generate a breadth-first expansion of $V_\Gamma(T_\emptyset)$ (modulo isomorphism) until the first 3 non-ϵ transitions occur along all branches. This allows deciding if a path labeled $\epsilon^*.\mathtt{e}.\epsilon^*.\mathtt{g}.\epsilon^*.\mathit{block}$ starting from the root exists in $V_\Gamma(T_\emptyset)$, and provides a procedure for testing whether $\Delta \models f$. □

Remark 6.10. By Lemma 2.5 (applied to effective simulations), the negative result of Theorem 6.9 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.

References

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

Abiteboul, S., Bourhis, P., and Marinoiu, A. G. B. 2009. The AXML artifact model. In *Time*.

Abiteboul, S., Bourhis, P., and Vianu, V. 2011. Comparing workflow specification languages: a matter of views. In *ICDT*. 78–89.

Abiteboul, S., Herr, L., and den Bussche, J. V. 1996. Temporal versus first-order logic to query temporal databases. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*.

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

Abiteboul, S., Segoufin, L., and Vianu, V. 2008. Static analysis of active XML systems. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*. Full paper in ACM TODS 34:4, 2009.

Abiteboul, S., Vianu, V., Fordham, B., and Yesha, Y. 2000. Relational transducers for electronic commerce. *Journal of Computer and System Sciences (JCSS) 61,* 2, 236–269. Extended abstract in PODS 98.

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

Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. 2003. XML with data values: typechecking revisited. *Journal of Computer and System Sciences (JCSS) 66,* 4, 688–727.

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

Arenas, M., Fan, W., and Libkin, L. 2002. Consistency of XML specifications. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*.

Bhattacharya, K., Gerede, C. E., Hull, R., Liu, R., and Su, J. 2007. Towards formal analysis of artifact-centric business process models. In *Int. Conf. on Business Process Management (BPM)*.

Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., and David, C. 2006. Two-variable logic on words with data. In *IEEE Symp. on Logic in Computer Scince (LICS)*.

BPEL. `http://bpel.xml.org/`.

Calvanese, D., Giacomo, G. D., Hull, R., and Su, J. 2009. Artifact-centric workflow dominance. In *IEEE Int. Conf. on Service-Oriented Computing and Applications (ICSOC)*. 130–143.

David, C. 2008. Complexity of data tree patterns over XML documents. In *Symp. on Mathematical Foundations of Computer Science (MFCS)*.

Demri, S. and Lazić, R. 2009. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic 10,* 3, 1–30.

Deutsch, A., Hull, R., Patrizi, F., and Vianu, V. 2009. Automatic verification of data-centric business processes. In *Int. Conf. on Database Theory (ICDT)*.

Deutsch, A., Sui, L., and Vianu, V. 2007. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences (JCSS) 73,* 3, 442–474.

Diekert, V. and Gastin, P. 2008. First-order definable languages. In *Logic and Automata: History and Perspectives*, J. Flum, E. Grädel, and T. Wilke, Eds. Vol. 2. 261–306.

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

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

FAN, W. AND LIBKIN, L. 2001. On XML integrity constraints in the presence of DTDs. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*.

FRITZ, C., HULL, R., AND SU, J. 2009. Automatic construction of simple artifact-based business processes. In *Int. Conf. on Database Theory (ICDT)*.

GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. 1995. An overview of workflow management: From process modeling to workflow infrastructure management. *Distributed and Parallel Databases 3*, 119–153.

GEREDE, C. E., BHATTACHARYA, K., AND SU, J. 2007. Static analysis of business artifact-centric operational models. In *IEEE Int. Conf. on Service-Oriented Computing and Applications (ICSOC)*.

GEREDE, C. E. AND SU, J. 2007. Specification and verification of artifact behaviors in business process models. In *IEEE Int. Conf. on Service-Oriented Computing and Applications (ICSOC)*.

HAREL, D. 1987. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program 8,* 3, 231–274.

HULL, R. 2009. Personal communication.

HULL, R., LLIRBAT, F., KUMAR, B., ZHOU, G., DONG, G., AND SU, J. 2000. Optimization techniques for data-intensive decision flows. In *Int. Conf. on Data Engineering (ICDE)*.

HULL, R., LLIRBAT, F., SIMON, E., SU, J., DONG, G., KUMAR, B., AND ZHOU, G. 1999. Declarative workflows that support easy modification and dynamic browsing. In *Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration*.

LIBKIN, L. 2004. *Elements of Finite Model Theory*. Springer.

MARTIN, D. ET AL. 2003. OWL-S: Semantic markup for web services, W3C Member Submission.

MCILRAITH, S. A., SON, T. C., AND ZENG, H. 2001. Semantic web services. *IEEE Intelligent Systems 16,* 2, 46–53.

MILNER, R. 1989. *Communication and concurrency*. Prentice-Hall, Inc.

MOK, W. AND PAPER, D. 2002. Using Harel's statecharts to model business workflows. *J. of Database Management 13,* 3, 17–34.

NARAYANAN, S. AND MCILRAITH, S. 2002. Simulation, verification and automated composition of web services. In *Int. World Wide Web Conf. (WWW)*.

NEVEN, F., SCHWENTICK, T., AND VIANU, V. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic 5,* 3, 403–435.

NIGAM, A. AND CASWELL, N. S. 2003. Business artifacts: An approach to operational specification. *IBM Systems Journal 42,* 3, 428–445.

SEGOUFIN, L. 2007. Static analysis of XML processing with data values. *SIGMOD Record 36,* 1, 31–38.

SPIELMANN, M. 2003. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences (JCSS) 66,* 1, 40–65.

VAN BENTHEM, J. 1976. Modal correspondence theory. Ph.D. thesis, Mathematish Instituut & Instituut voor Grondslagenonderzoek, Univ. of Amsterdam.

VAN DER AALST, W. 2004. Business process management demystified: A tutorial on models, systems and standards for workflow management. In *Lectures on Concurrency and Petri Nets*.

VAN DER AALST, W. M. P. 1998. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers 8,* 1, 21–66.

VAN DER AALST, W. M. P. AND TER HOFSTEDE, A. H. M. 2002. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In *Proc. of the Fourth Int. Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, 2002*.

VARDI, M. Y. 1996. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata, volume 1043 of LNCS*. Springer-Verlag, 238–266.

WANG, J. AND KUMAR, A. 2005. A framework for document-driven workflow systems. In *Int. Conf. on Business Process Management (BPM)*. 285–301.