On the dynamics of active documents for distributed data management

Pierre Bourhis

UNIVERSITÉ
PARIS-SUD 11

Thèse de doctorat en informatique

Étude de la dynamique des documents actifs pour la gestion d'information distribuées

On the dynamics of active documents for distributed data management

Pierre BOURHIS

11 Février 2011

Jury

Serge ABITEBOUL	DR INRIA Saclay	(directeur)
Michael BENEDIKT	Prof. Univ. Oxford	(rapporteur)
Albert BENVENISTE	DR INRIA Rennes	(rapporteur)
Nicole BIDOIT	Prof. Univ. Paris Sud	
Anca MUSCHOLL	Prof. Univ. Bordeaux 1	
Victor VIANU	Prof. Univ. San Diego	

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Étude de la dynamique des documents actifs pour la gestion d'information distribuées
On the dynamics of active documents for distributed data management

Pierre BOURHIS

Résumé

L'un des principaux problèmes que les applications Webs doivent gèrer aujourd'hui est l'évolutivité des données. Dans cette thèse, nous considèrons ce problème et plus précisément l'évolution des documents actifs. Les documents actifs sont documents XML pouvant évolués grâce à l'activation d'appel de services Web. Ce formalisme a déjà été utilisé dans le cadre de la gestion d'information distribuée. Les principales contributions de cette thèse sont l'étude théorique de différentes notions pour l'implémentation de deux systèmes gèrant des applications manipulant des flux de données et des applications de type workflow. Dans un premier temps, nous étudions des notions reliées à la maintenance de vues sur des documents actifs. Ces notions sont utilisées dans l'implémentation d'un processeur de flux de données appelé Axlog widget manipulant des flux à travers un document actif. La deuxième contribution porte sur l'expressivité de différents formalismes pour contraindre le séquencement des activation d'un document actif. Cette étude a été motivée par l'implémentation d'un système gérant des workflows focalisés sur les données utilisant les documents actifs, appelé Axart.

Abstract

One of the major issues faced by Web applications is the management of evolving of data. In this thesis, we consider this problem and in particular the evolution of active documents. Active documents is a formalism describing the evolution of XML documents by activating Web services calls included in the document. It has already been used in the context of the management of distributed data [Abiteboul 08a]. The main contributions of this thesis are theoretical studies motivated by two systems for managing respectively stream applications and workflow applications. In a first contribution, we study the problem of view maintenance over active documents. The results served as the basis for an implementation of stream processors based on active documents called Axlog widgets. In a second one, we see active documents as the core of data centric workflows and consider various ways of expressing constraints on the evolution of documents. The implementation, called Axart, validated the approach of a data centric workflow system based on active documents.

Mots clefs : XML, documents actifs, satisfiabilité, pertinence, workflow

Keywords: XML, active documents, satisfiability, relevance, workflow

À l'exception de l'annexe C, qui propose un résumé de la thèse , cette thèse est rédigée en anglais.

With the exception of Appendix C, that is a summarize of the thesis, this thesis is written in English.

Introduction

One of the major issues faced by Web applications is the management of evolving of data. In this thesis, we consider this problem and in particular the evolution of active documents. Active documents is a formalism describing the evolution of XML documents by activating Web services calls included in the document. It has already been used in the context of the management of distributed data [Abiteboul 08a]. The main contributions of this thesis are theoretical studies motivated by two systems for managing respectively stream applications and workflow applications. In a first contribution, we study the problem of view maintenance over active documents. The results served as the basis for an implementation of stream processors based on active documents called Axlog widgets. In a second one, we see active documents as the core of data centric workflows and consider various ways of expressing constraints on the evolution of documents. The implementation, called Axart, validated the approach of a data centric workflow system based on active documents.

In a first part, we focus on streaming applications. The Web includes a large number of sources consisting of XML streams such as news or Blog feeds. Many Web pages are simply aggregations of news feeds. At the heart of such pages, one finds stream querying. We present a formal model, called Axlog, that captures simple queries over streaming sources. Our approach is in the spirit of view maintenance over active documents. Our main contribution, published in [3], is a study of two theoretical notions: satisfiability and relevance. We briefly outline an algorithm to maintain a view over document that includes input streams, see [4]. The algorithm uses these theoretical notions in order to combine together database techniques: optimization evaluation techniques for datalog queries, view maintenance techniques, stream processing techniques for XML ,and filter techniques. The Axlog model is supported by the system P2PMonitor, that is demonstrated in [9]. P2PMonitor is a peer to peer system which monitors other peer to peer systems by managing XML stream queries. The system P2PMonitor and the view maintenance algorithm have been presented in detail in the thesis of Bogdan Marinoiu [Marinoiu 09].

In a second part, we address the problem of sequencing interactions between Web applications. E-commerce Websites are a good example of applications where sequencing is crucial. The interactions between the users and the applications are constrained in order to conform to a workflow. Several workflow languages have been introduced, such as BPEL. However, most of the languages focus on the sequencing of the actions rather than on the data used in the process. New kinds of workflow languages more focused on data, called data-centric workflows, have recently been introduced. We present the AXML Artifact model, [5], inspired by the Business Artifact model, a data-centric workflow language introduced by IBM. Our main contribution, published in [2], studies and compares different ways of expressing the sequencing of the operations based on different paradigms including automata, pre- and post-conditions for operations, or temporal logic. We briefly describe a system [8] implementing a portion of the AXML Artifact model.

The thesis is organized in two parts. The first deals with streaming applications and the second with sequencing. In each part, we use the same organization. After an overview of the part, we discuss the related work. We then present the model and study theoretical issues for the particular model. Finally, we briefly discuss the implementation work based on these theoretical studies.

The first part is composed of three chapters: Chapter 1 presents the related work. Chapter 2

describes the model and the study of the two key notions of *satisfiability* and *relevance* in the context of the Axlog model. Chapter 3 describes the algorithm proposed to efficiently implement applications based on the Axlog model and the system P2PMonitor supporting them.

The second part is composed of three chapters. Chapter 4 presents the related work. Chapter 5 describes the core of the model and the study of constraints specifying the evolution of active documents. Finally Chapter 6 discusses some extensions of the model and an implementation.

Detailed proofs are provided in the appendix. Appendix C is a resume into French of the thesis.

Part I.

Maintenance of Views over Active Documents

Overview of Part I

Many Web applications are based on dynamic interactions between Web components exchanging flows of information. Such a situation arises for instance in mashup systems [Ennals 07] or when monitoring distributed autonomous systems [Abiteboul 07]. This is a challenging problem that has recently generated a lot of attention; see Web 2.0 [O'Reilly]. Starting from datalog and Active XML technologies, we introduce a novel model, Axlog, for capturing interactions between Web components and show how it can be supported efficiently. An *Axlog widget* uses an active document interacting with the rest of the world via streams of updates. Its input streams specify updates the document (in the spirit of RSS feeds), whereas its output streams are defined by queries on the document. More precisely, the output stream represents the list of update requests to maintain the view for the query. The queries we consider here are tree-pattern queries with value joins (and a template to produce an XML result). Our data model and queries may include a time dimension, an essential feature for such a setting. The crux of the support of Axlog widgets is the maintenance of views corresponding to queries over the active documents. We exploit an array of known technologies for datalog optimization techniques such as MagicSet, view maintenance optimizations such as Differential technique and efficient XML filtering. The novel optimization technique we propose is based on two fundamental new notions: relevance (different than that of MagicSet), satisfiability for active documents.

First, we present the core of the model where the updates are only insertions and the queries are tree pattern queries. We introduce and study two fundamental concepts in this setting, namely, satisfiability and relevance. Some fact is *satisfiable* for an active document and a query if it has a chance to be in the result of the query in some future state. Given an active document and a query, a call in the document is *relevant* if the data brought by this call has a chance to impact the answer to the query. We analyze the complexity of computing satisfiability in our core model, and for extensions. In the core of the model, the only updates are insertions. In the extensions, the active document may be associated to a schema, the updates may also be deletions, the queries may contain some negations and the model and the queries may include a time dimension. We also analyze the complexity of computing relevance for the core model.

The Axlog system is the core of the P2PMonitor system. We briefly present the system and the integration of Axlog widgets in it. We briefly explain how satisfiability and relevance are used in the view maintenance algorithm. To summarize, Axlog widgets can be used to support a number of tasks in distributed environments such as stream processing. The platform supporting Axlog widgets has been demonstrated in [9] using a supply chain application [Kapuscinski 04]. It is used in a new version of a P2P monitoring system, P2PMonitor [Abiteboul 07]. The implementation uses an efficient view maintenance algorithm [4]. This algorithm is notably based on satisfiability and relevance [3].

The first part is composed of three chapters: Chapter 1 presents the related work. Chapter 2 describes the model and the study of the two key notions of *satisfiability* and *relevance* in the context of the Axlog model. Chapter 3 describes the algorithm proposed to efficiently implement applications based on the Axlog model, and the system P2PMonitor supporting them.

Chapter 1.

Related Work

This chapter presents related works for the first part. It is organized as follows. First, we briefly present the main streams systems that have already been implemented. Then, we present the techniques for query evaluation, we use or that are related on our work. The problem of view maintenance has already been studied in depth. We mention some key techniques. The notion of satisfiability for queries over trees has already been studied in other contexts that we present here. Moreover, the notion of satisfiability of a query over active documents is related to the problem of querying incomplete informations that we also mention. Finally, one of the extensions of Axlog widget is concerned with typing. We discuss typing notions related to the one we use in this thesis.

1.1. Stream processing systems

1.1.1. Relational streams systems

Data stream processing has been intensively studied, in particular for the relational model, e.g. the Borealis [Abadi 05], Aurora [Abadi 03] and STREAM [Motwani 03] systems. The main focus of these systems is to deal with very intensive stream sources and a very large number of queries. Their approach is typically to rewrite the query into an efficient plan of stream processors communicating together. Most of the optimizations are based on using nonblocking operators. A blocking operator is an operator that has to know the entire definition of the streams to fully process them. For example, the join operator is blocking.

1.1.2. XML streams systems

In the XML stream processing field, lots of works focus on stream filtering, e.g [Diao 02, Green 03]. XML stream filters are generally based on automata, either non-deterministic [Diao 02], or deterministic [Green 03]. These approaches scale very well with the number of queries on a stream by aggregating the different filter into a single one.

Several XQuery processors for XML data streams have been proposed, e.g. [Florescu 03, Koch 04], as well as distributed systems that handle streams e.g. StreamGlobe [Kuntschke 05, Stegmaier 04]. Some works, e.g. [Fernández 07], are blending stream processing with optimization techniques for XML databases, however in a quite different setting than the one we consider here.

1.2. Query evaluation

1.2.1. Datalog

Datalog [Abiteboul 95] is a declarative language to query relational data. In this part, we intensively use the fact that tree-pattern queries can be rewritten into a datalog program as shown in [Gottlob 02,

Gottlob 05, Miklau 04]. We use datalog and benefit from known optimization techniques such as Magic Set [Beeri 91] (or QSQ [Vieille 89]).

To compute the satisfiability of queries over an active document, we use constraint query languages [Kanellakis 95], that is an extension of datalog to manipulate constraints.

In [Ronen 07], datalog is extended with XPath predicates; the evaluation is not incremental.

1.2.2. AXML

Query evaluation for active documents is studied in [Abiteboul 04a]. Their goal is to know which functions have to be called to answer the query. The context is essentially different since the functions are non-stream and the incremental maintenance is not considered. In the context of distributed query optimization, an optimizer for AXML, called Optimax [Abiteboul 08b], has been developed. In this context, a recent study has been done on the equivalence of AXML documents [Abiteboul 11].

1.3. View Maintenance

1.3.1. Incremental View Maintenance

Our work is based on previous works on incremental view maintenance for relational databases [Blakeley 86b, Ceri 94, Gupta 95, Gupta 93]. These works proposes an optimization method typically by rewriting the query in order to compute only the new facts. Some auxiliary structures, such as the number of times that a fact is derived, may have to be maintained if the deletions are considered.

Incremental view maintenance for a graph semistructured data is studied in [Abiteboul 98]. Some recent works have addressed the issue of incremental maintenance of no-join XPath views over trees [Onizuka 05, Sawires 05, Björklund 09]. In particular, the work [Björklund 09] gives theoretical bounds over the structures used to maintain Boolean no-join tree-pattern queries. The techniques are based on automata theory. The maintenance of XQuery views is studied in [Foster 08] but without data streams and with data fully residing in memory.

1.3.2. Relevance

The relevance of an update for a query/view has been intensively studied in relational databases, e.g., [Blakeley 86a, Levy 93, Calì 08]. The fact that a stream is relevant, is related to the notion of critical tuple for a conjunctive query presented in [Miklau 07].

The relevance of updates for XML trees have been studied in [Benedikt 10, Benedikt 09]. The updates and queries are expressed in a large subset of XQuery, which is more powerful than our tree-pattern queries. In [Benedikt 09], the relevance of updates is defined only from the schema of the document. In [Benedikt 10], the relevance of updates is based on the notion of a set of nodes that may impact the query if they are affected by an update. Because of the complexity of the updates and the query, the computation of this set is non primitive recursive.

In active document contexts, function call Id relevance has also been studied [Abiteboul 04a, Abiteboul 04b, Abiteboul 06]. The notions of relevance we introduce below (relevance and axlog-relevance) are both more refined than previous notions such as lazy relevance.

1.4. Satisfiability and containment of queries

The notion of satisfiability of queries over trees have been studied for different formalisms both for static and for evolving trees.

1.4.1. Satisfiability of tree-pattern queries over static trees

In our study of satisfiability, we use some previous results on tree-pattern query satisfiability and containment possibly under constraints such as DTD [Benedikt 08, Björklund 08, David 08, Figueira 09, Miklau 04]. All these works are about trees with multiset semantics. As we demonstrated in Section 2.4, the semantics of reduced trees introduces some subtleties. However, we explain in Appendix A.3 how to use previous works in our setting.

First, remark that tree-pattern queries are always satisfiable if the trees do not have to respect type constraints. The query containment of Boolean tree-pattern no-join queries has been studied in [Miklau 04]. The complexity is co-NP-complete.

The satisfiability and query containement of XPath queries are studied in [Benedikt 08, Figueira 09]. XPath queries are less powerful than tree pattern queries in particular because they cannot express arbitrary joins (see Figure 1.1). In [Benedikt 08], the question of satisfiability of positive XPath under DTD and containment of XPath queries is studied. The satisfiability of a positive Xpath query under a DTD is NP-complete. In the context of XPath, [Figueira 09] shows that the satisfiability of an XPath query with negation over unordered trees is decidable. Its complexity is EXPTIME. It gives an upper-bound for query containment of Xpath under DTD.

Satisfiability and containment of conjunctives queries over trees is considered in [Björklund 08]. Conjunctive queries include tree-pattern queries. The satisfiability of a conjunctive queries under DTD is NP-COMPLETE and the containment of two conjunctive queries is 2-EXPTIME.

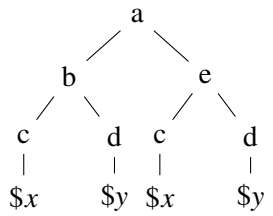


Figure 1.1.: An arbitrary join tree-patern query

Satisfiability of Boolean combination of tree-patterns under DTD is studied in [David 08]. This problem is undecidable in general. The problem becomes decidable for bounded depth trees and the complexity is NEXPTIME. There are subtleties about the semantics of tree pattern queries in [David 08], inequalities constraints between variables are allowed and the valuation between a tree-pattern query and a tree has to be injective, that is not our semantics. However, most of the results can be adapted by removing these hypotheses. In particular, the satisfiability of Boolean combination of tree patterns under DTD is also undecidable in our model.

Satisfiability for evolving trees There have been works on the verification of temporal properties for active documents. [Ma 08] also studies active document satisfiability for tree-pattern queries. However, it deals with ordered trees, which is, as mentioned in the introduction, a

much more complex issue. The paper is rather imprecise and the results seem to contradict well-known results [Muscholl 04]. [Abiteboul 04b] studies reachability for monotone AXML systems. [Abiteboul 09, Genest 08, Genest 10] study a rather general class of non monotone AXML systems and a very large class of temporal formulas. Our model is in many aspects more limited than those used in these previous works. This is the price to pay to obtain PTIME data complexities as in our framework. However, it should be noted that the setting we consider allows unbounded runs (which is not the case in [Abiteboul 09]) and infinite data values (which is not the case in [Genest 08]). [Genest 10] proposes a more general formalism than ours but the known complexity of the satisfiability of a query is not primitive recursive.

1.5. Incomplete information

1.5.1. Incomplete relational databases

First, the problem of satisfiability we study is related to the problem of querying an incomplete database, e.g. [Grahne 91, Imielinski 82, Levy 96]. The representation of satisfiable answers we use is related to the C-tables introduced in [Imielinski 84]. A C-table can be seen as a table of generalized tuples. A C-table represents the set of instances that can be obtained by mapping a valuation of the variables appearing in the C-table to it. This notion is more precise than the notion of satisfiability introduced in our context. Indeed, our definition of satisfiability defines the union of all the possible tuples that may be obtained by updates of the active document.

1.5.2. Trees and incomplete information

Active documents can be seen as trees with incomplete information, where function call Ids represent incomplete information. The typed active documents we use are related to the incomplete trees used in [Abiteboul 06]. They use trees with specific labels associated to a DTD. These labels represent the existence of subtrees that satisfies the associated DTD. In this paper, the authors explain how to refine an unknown XML database by querying it. In particular how to check if a query may be satisfied by the unknown database by using incomplete information. The DTD and queries are slightly different from ours. Their DTD can only express the possibility of the existence of a subtree labeled by a label a (and not a precise number). They can express inequalities constraints on values. In the same way, their queries based on tree-patterns cannot express joins but can express inequalities of values to some constant. Another main difference comes from the set semantics of trees used in our model vs the bag semantics in [Abiteboul 06] .

1.6. Type

The DTD formalism for unordered and reduced trees has already been introduced in [Abiteboul 09]. However a complete study of this formalism was not done in [Abiteboul 09]. To our knowledge, the notions of reduced tree and unordered trees have not been studied simultaneously in the context of regular languages. However, these notions have been considered separately as summarized next.

1.6.1. Unordered trees

Our notion of unordered DTD is inspired by other formalisms proposed for unordered trees. One of the most general typing formalisms for unordered trees has been considered in [Seidl 03]. It

introduces a kind of tree automaton where constraints are expressed by Presburger formulas. In particular, they show that for a particular formalism emptiness is NP-COMPLETE.

1.6.2. Reduced trees

The notion of reduced trees is related to the tree automata with sibling constraints for unranked trees [Wong 07, Löding 09]. The constraints expressible in this formalism allow imposing equalities and disequalities over sibling trees. They show that emptiness is decidable for such automata.

Chapter 2.

Satisfiability and relevance for queries over active documents

2.1. Introduction

In this chapter, we introduce the core of the Axlog model, i.e active documents with queries that define a view on them. An active documents is a tree with a set semantics for children of a node and interacting with the rest of the world via streams of updates. For the queries, we use tree-pattern queries with joins whose answers are tuples of bindings of the variables in the pattern. Such a document with queries (i.e. views) defined on it is what we call an *Axlog widget*. The term Axlog results from the marriage between Active XML (AXML for short) [Abiteboul 08a] and datalog. The input streams of an Axlog widget specify updates to the document (in the spirit of RSS feeds). In most of the chapter, the focus is on input streams where the updates are only insertions, although we do consider also deletions. An output stream is defined by a query on the document. More precisely, it represents the list of update requests to maintain the view of the query.

Our main contribution is a study of two novel notions for active documents, satisfiability and relevance. First, we say that some fact is *satisfiable* for an active document and a query if it has a chance to be in the result of the query in some future state. In the spirit of the evaluation of tree-pattern queries using datalog [Gottlob 02], we show how to evaluate query satisfiability in datalog, so in PTIME in the size of the document. Note that the number of satisfiable tuples may be infinite. We use a finite representation based on tuples with variables for the set of satisfiable tuples. To handle these representations, we use the constraint query language CQL [Kanellakis 95].

We also study satisfiability (for a document and a query) for extensions of the model. First we introduce typing. We consider typing for both the documents [Comon 97, DTD] and the data on the input streams [WSDL]. Since we use set semantics, we adapt DTDs to ignore the ordering of siblings. We show how to evaluate satisfiability for documents constrained by unordered-DTDs. Then we consider a number of nonmonotonic features, like deletions, terminating calls, negation in queries. In particular, we see that negation rapidly leads to undecidability of satisfiability. Finally we consider temporal queries, a most useful feature in the context of active documents, e.g., for monitoring. We extend the model with time and show how to evaluate satisfiability building on constraint query languages.

The second key notion we study is relevance. Given an active document and a query, a call in the document is *relevant* if the data brought by this call has a chance to impact the answer to the query. This is in the spirit of data relevance in MagicSet [Beeri 91] and lazy-relevance in [Abiteboul 04a]. Relevance of function calls has also been studied in [Abiteboul 04b]. Some works on view maintenance also discuss relevance of updates, as in [Blakeley 86a, Abiteboul 06]. We show how to evaluate relevance in PTIME in the size of the data. The combined complexity is high and the PTIME algorithm is too expensive for practical purposes. We propose a weaker condition

namely *axlog-relevance*, that is easier to verify.

The work presented here has been used in the implementation of a system supporting Axlog widgets outlined in Chapter 3.

We want to stress the fact that we consider only unordered trees (set semantics for the children of a node). Results in [Muscholl 04] indicate that document satisfiability for ordered trees and types specified by DTDs is much more complicated.

The chapter is organized as follows. In Section 2.2, we formalize the model. In Section 2.3, we study satisfiability. We consider types in Section 2.4 and other extensions of the model in Section 2.5. Section 2.6 is about relevance. The proofs are given in appendix.

2.2. The model

In this section, we define the data structure (active documents) and the query language (tree-pattern queries) that are used for defining the Axlog model. We define Axlog systems. We introduce here the core model. We will consider a number of extensions in Sections 2.4 and 2.5.

2.2.1. Definitions

We assume the existence of some infinite alphabets $\mathcal{I}$ of node identifiers, $\mathcal{L}$ of labels, $\mathcal{C}$ of (function) call Ids, and $\mathcal{V}$ of variables. We do not distinguish here between XML data, attributes and labels, i.e., our labels are meant to capture these three notions. We use the symbols n, m, p for node identifiers, $a, b, c...$ for labels, $?f, ?g, ?h...$ for call Ids, possibly with sub and superscripts, and $\$x, \$y, \$z...$ for variables. We consider active documents in the style of AXML [Abiteboul 08a, Axml]. Such documents may be viewed as abstractions of XML documents including calls to external resources, e.g., Web services or user inputs.

Definition 2.1 (Active document)**.** *An active document* is a pair (t, λ) where (1) t is a finite binary relation that is a tree* with $nodes(t) \subset \mathcal{I}$; (2) λ is a labeling function over $nodes(t)$ with values in $\mathcal{L} \cup \mathcal{C}$; and (3) the root and each node that has a child are labeled by values in $\mathcal{L}$ (so only leaves may be labeled by values in $\mathcal{C}$). We also impose that: (4) no call Id occurs more than once in an active document.

A *(data) forest* is a finite set of documents and of trees consisting of a single node with a label from $\mathcal{C}$.

Remark 1. In AXML, a *function call node* has children denoting the parameters of the call. We consider in this paper that calls have already been made and a *function call node*, labeled with a call Id, is just a marker to indicate where the results of the call should go, i.e., as siblings of this node. Observe that, therefore, function call nodes do not have children, so there is no nesting of such nodes.

Four examples of documents are given in Figures 2.1 and 2.2. The last document of Figure 2.2 presents the members of the Webdam team. Calls to the personnel database feed the document namely ?researcher and ?phd. In a standard database manner, we use in this paper a set semantics for the children of a node. Two active documents (t, λ), (t', λ') are *isomorphic* if they differ in their node identifiers only. In the following, we will consider that all documents are *reduced*, i.e., that they don't include a tree node with two isomorphic subtrees. Clearly, each document can be

*The trees that we consider here are unordered and unranked.

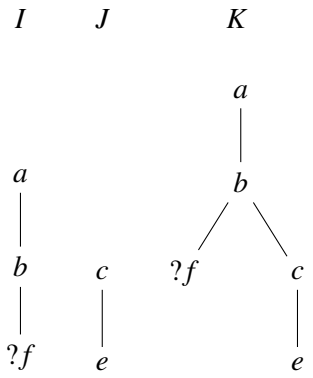


Figure 2.1.: Updating of an active document

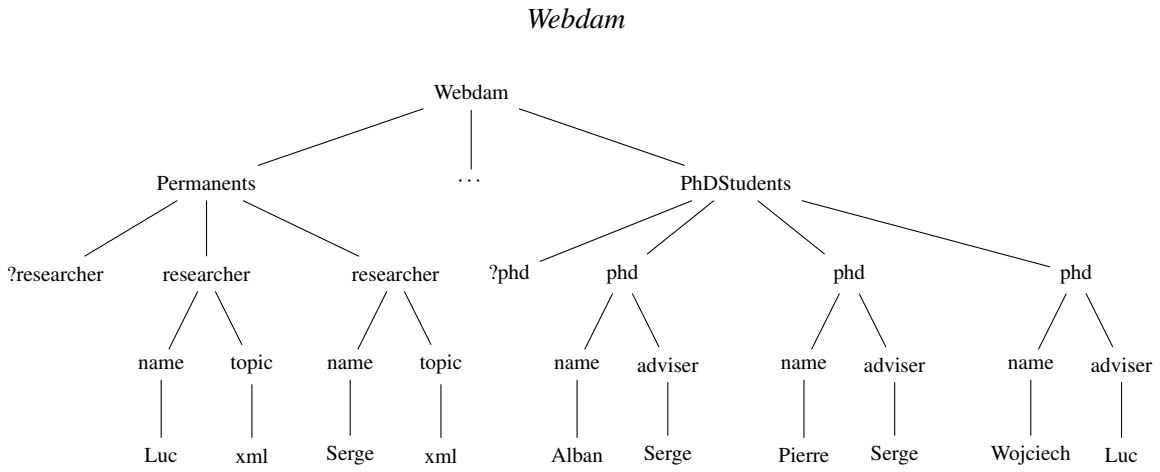


Figure 2.2.: An example of active documents

reduced by eliminating duplicate isomorphic subtrees, and the result is unique up to isomorphism. These notions are lifted to forests in the straightforward manner.

Calls can be seen as subscriptions (to some services) and are meant to receive streams of updates. Trees evolve in time by receiving results of such update requests from the services called in them. To simplify, we first consider that (1) the incoming flow of updates consists only of insertions, and (2) these flows return data not containing new calls. More precisely, an *insertion* for an active document I is an expression $add(?f, J)$ where $?f$ is a call occurring in I and J a "passive" active document (i.e., it contains no calls). Let I be an active document, $add(?f, J)$ an update to I and n the node of I labeled $?f$. The *result* of applying $add(?f, J)$ to I, denoted $add(?f, J)(I)$, is the active document obtained from I by adding, as a sibling of n, a fresh copy* J' of J. For instance, for I, J, K as in Figure 2.1, $K = add(?f, J)(I)$. The active document obtained from I by applying a sequence ω of inserts is denoted $\omega(I)$. To generalize, we also see an expression $add(?f, \{J_1, ..., J_n\})$ as an update. Applied to some active document I, it has the same effect as the sequence $add(?f, J_1);...; add(?f, J_n)$ of updates (for some ordering of the updates in the set). Observe that the order of application of these updates is irrelevant. This will no longer be true when we consider extensions of the model. Also note that, by definition of active documents, a tree consisting of a single function call node is not a document. This is ruled out because a call may request the insertion of a set of trees, so yield a forest.

Constraint (4) in the definition of active document may seem arbitrary. We justify it next. We can extend the definition of updates to allow the multiple occurrences of a call Id. When such a call returns some data, isomorphic copies of this data are inserted in the document in various places corresponding to occurrences of the call. This introduces some nonregularity (in the sense of regular trees). To see that, consider the set of documents that can be reached from the document $r[a[?f]][b[?f]]$.

The queries considered in our model are tree-pattern queries. Examples are given in Figure 2.3. The single lines indicate a parent relationship, and the double lines an ancestor relationship. The $-variables may match any label. A variable $\$x$ is requested to be in the result if marked by a "+". The result therefore consists of tuples over the variables marked with "+". The variables that are not marked by a "+" are existentially quantified. So a query, in which there is no "+" mark, is a boolean query. Query q_1 is a Boolean query, and the other three queries return binary relations over $\$x$ and $\$y$. We consider only equality joins for now. We will introduce other comparators further on. Formally, we have:

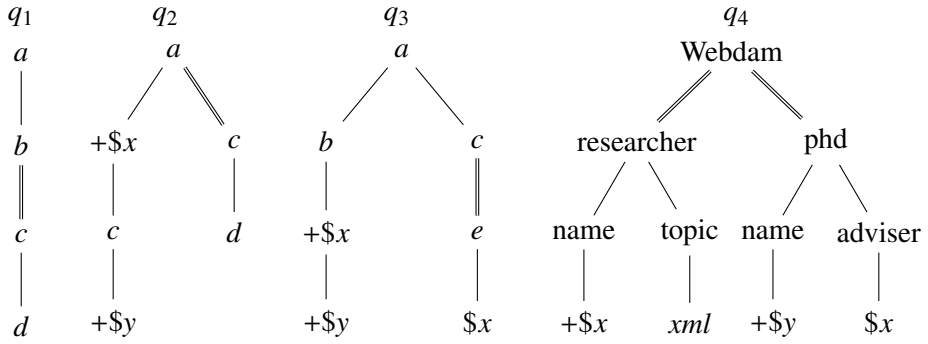


Figure 2.3.: Examples of queries

*The copy is isomorphic to J and its nodes are disjoint from the nodes of I.

Definition 2.2. (tree-pattern query) A *(tree-pattern) query* q is an expression $(E_/, E_{//}, \lambda, \pi)$ where: (i) $E_/$, $E_{//}$ are finite, disjoint subsets of $\mathcal{I} \times \mathcal{I}$, and $(E_/ \cup E_{//})$ is a tree; (ii) The labeling function λ maps $nodes(q)$ to $\mathcal{L} \cup \mathcal{V}$; and (iii) projection π is a subset of $nodes(q)$ with labels in $\mathcal{V}$, where $nodes(q)$ is the set of nodes in $E_/ \cup E_{//}$.

The semantics of queries is defined as follows.

Definition 2.3. (Semantics of TPQ) Let $q = (E_/, E_{//}, \lambda, \pi)$ be a query and $I = (t', \lambda')$ a document. A *valuation* ν from q to (t', λ') is a mapping from $nodes(q)$ to $nodes(t')$ that is:
(i) Root-preserving: $\nu(root(q)) = root(t')$.
(ii) Parent/descendant preserving: For each $(p, p') \in E_/$, $\nu(p)$ is a parent of $\nu(p')$ in t'; and for each $(p, p') \in E_{//}$, $\nu(p)$ is an ancestor of $\nu(p')$ in t'.
(iii) Label-preserving: For each $p \in nodes(q)$, if $\lambda(p) \in \mathcal{L}$ then $\lambda'(\nu(p)) = \lambda(p)$, otherwise $\lambda'(\nu(p)) \in \mathcal{L}$.
(iv) Join-obeying: If $\lambda(p) = \lambda(p') \in \mathcal{V}$, then $\lambda'(\nu(p)) = \lambda'(\nu(p'))$.
The *result* $q(I)$ is the relation $\{\lambda'(\nu(\pi)) \mid \nu \text{ a valuation}\}$.

If there is no variable occurring more than once, the query is said to be a *no-join* query. If π is empty, the query is said to be a *Boolean* query. Its result is then either the empty set (false) or the set containing the empty tuple (true). For a Boolean query q, if $q(I)$ is true, we say that I *satisfies* q, denoted $I \models q$. For a non-Boolean query q, using standard notation, we denote the fact that a tuple u is a result, i.e. $u \in q(I)$, by $I \models q(u)$.

In general, we can use non-recursive datalog (nrec-datalog for short) to compute the answers to a query in the style of [Gottlob 02, Miklau 04]. We assume, in a standard manner, that the document is represented in a relational database using the extensional relations *root, child*, *descendant, label* with their standard meaning ; in particular, $label(a, x)$ holds if the node with identifier x is labeled by $a \in \mathcal{L}$. The representation also uses a unary relation, namely *function*, with the semantics that $function(x)$ holds if the label of node x is in $\mathcal{C}$, i.e., x is a function call node. We construct by recursion the datalog program P_q that computes $q(I)$ given I using the programs corresponding to its subqueries. The program has one relation p for each node p of q. The relation for the root of q defines the answer. Observe that the datalog program needs to carry along labels if they are potentially in the result or can potentially be joined to other labels. Observe also that the function call nodes play no role for computing the answers to the query. Details are omitted. Efficient algorithms for evaluation of tree-pattern and XPath queries can be found in [Gottlob 05].

2.2.2. Axlog Systems

We now consider Axlog systems where a call Id may correspond to a subscription to a query over some active document of the system. This introduces recursion in the evaluation.

We want queries producing trees whereas the queries so far only produce tuples. For that, we define queries with template that consist of pairs (q, t) where q is a query and a template t is a tree where the labels are constants and variables occurring in q. A result for such a query is obtained by replacing the variables in the template t by the valuation of the variables in q as given by a result tuple.

Formally, an *Axlog system* $\mathcal{S}$ is composed of:

- A finite set of *Axlog widgets*, where an Axlog widget is a pair $(d, (q, t))$ where d is a document and (q, t) a query with template, such that no call Id occurs twice.

- A function ξ over the calls Ids occurring in the documents such that for each $?f$, $\xi(?f)$ is either a specific symbol $\top$ (the call is external) or some $(d,(q,t))$ in the system (the call is internal).
- For each document d, a queue of updates not yet treated denoted, B_d. These queues are originally empty.

The evolution of an Axlog system is defined as follows:

1. an update may come from outside, i.e some $add(?f, K)$ for each external call occurring in some document d. This update is added to the queue B_d.
2. the first update in a queue of some document arbitrarily chosen, say $(d,(q,t))$, is applied to the document. The queries associated to a document produce new updates, i.e. trees not appearing previously in the answer of the query, that are propagated to the corresponding queues.

An Axlog system captures exactly the behaviors of a set of Axlog widgets implemented in P2PMonitor.

2.3. Satisfiability

We are interested in the evolution of *views* over such documents, defined by tree-pattern queries. First, given a document, we want to know if a Boolean query holds in some reachable state. Similarly, we are concerned with determining whether a tuple belongs to the view in some reachable state of the document. These notions are related to that of coverability in dynamic systems [Finkel 01]. To investigate these issues, we introduce and study the notion of query satisfiability for a document.

Definition 2.4. Given a document I and a query q, a tuple u is *satisfiable* for (I,q) if $u \in q(\omega(I))$ for some (possibly empty) sequence ω of insertions. We say that a Boolean query q is *satisfiable* for I, if for some ω, $\omega(I) \models q$, i.e., the tuple () is satisfiable for (I,q). This is denoted by $I \models \diamond q$. Clearly, if $I \models q$, then $I \models \diamond q$.

For example, the tuple () is satisfied for (I_1, q_0) of Figure 2.4, denoted by $I_1 \models q_0$. Similarly, for q_0, I_2, I_3 in Figure 2.4, $I_2 \not\models \diamond q_0$ and for $I_3 \models \diamond q_0$. For I_3, q_0 does not hold but $?f$ may bring some node labeled c to make q_0 hold. Observe that this leads to some form of 3-valued logic where a tuple may be true, false for now but possibly true in some future, or false forever. This notion is interesting in its own right. For instance, consider in the context of the supply chain application of [Kapuscinski 04] a query that selects the mail orders that completed successfully. One may want to know which mail orders still have a chance to complete successfully even though they are not part of the query result yet.

We also use a datalog program to compute the set of satisfiable tuples for (I,q). We assume that the document is represented in a relational database using the extensional relations *root, child, descendant, label, function, time* with the standard meanings. In particular, $label(a,n)$ (respectively $function(n)$) holds if the node with identifier n is labeled by $a \in \mathcal{L}$ (respectively $?f \in \mathcal{F}$). As previously mentioned, in the computation of satisfaction, one carries along the bindings of the variables that may occur in the result or be joined to other labels. For satisfiability, this is more intricate since parts of the bindings may be brought by future inserts and may still be unavailable.

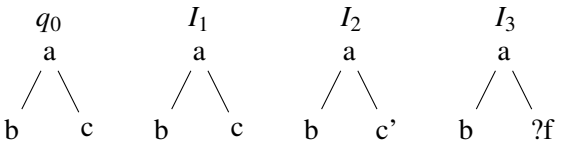


Figure 2.4.: A query and some active documents

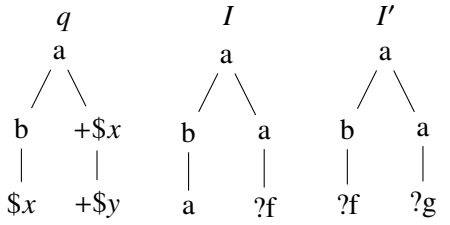


Figure 2.5.: A query and some active documents

In particular, the set of successful bindings for satisfiability may be infinite. (So, there is typically no reachable instance that contains all the complete satisfiable tuples since there may be infinitely many such tuples and instances are finite). To overcome this difficulty, we use generalized n-tuples and the constraint query language CQL [Kanellakis 95], an extension of datalog with constraints. We only need for now equality constraints between variables and possibly constants.

Definition 2.5. A *generalized n-tuple* is a pair $(u, \mathcal{C})$ where u is a tuple of variables and $\mathcal{C}$ is a set of satisfiable constraints of the form $\$x = \$y$ or $\$x = a$ for some a in $\mathcal{L}$.

A generalized tuple $(u, \mathcal{C})$ is a finite representation for a possibly infinite set of (complete) n-tuples, i.e., the set of tuples $\theta(u)$ for some instantiation θ of the variables satisfying $\mathcal{C}$. We say that a generalized n-tuple $(u, \mathcal{C})$ is *satisfiable* for (I, q) iff for each instantiation θ of u satisfying $\mathcal{C}$, $\theta(u)$ is satisfiable for (I, q). Consider Figure 2.5. One can verify that, for instance, $(\$x, \$y, \$x = a)$ and $(\$x, \$y, \$x = a \wedge \$y = c)$ are satisfiable for (I, q) and for (I', q). The generalized tuple $(\$x, \$y, \varnothing)$ is satisfiable for (I', q) but not for (I, q).

The previous discussion motivates the following auxiliary notion. Let $(u, \mathcal{C})$ and $(u', \mathcal{C}')$ be two generalized n-tuples over the same set of attributes. Then $(u', \mathcal{C}') \sqsubseteq (u, \mathcal{C})$, $(u, \mathcal{C})$ *more general than* $(u', \mathcal{C}')$, iff for each instantiation θ' of u' satisfying $\mathcal{C}'$, there is an instantiation θ of u satisfying $\mathcal{C}$ and $\theta(u) = \theta'(u')$. (This corresponds to the existence of a homomorphism from $(u', \mathcal{C}')$ to $(u, \mathcal{C})$.)

Let $q = (E_/, E_{//}, \lambda, \pi)$ be a tree-pattern query. We now sketch the construction of a program P_q that computes the satisfied tuples for an active document. This program is used to build the program $\widehat{P}_q$ that computes the generalized tuples for satisfiability. Program P_q has two different kinds of intensional relations:

- One intensional relation for each node p, denoted p. The relation p has an arity equal to the number of different variables appearing in the subtree rooted by p and appearing in π or in another part of the tree plus one.

- One intensional relation q. This is the output relation and has an arity equal to the arity of π.

Given an active document I, a tuple $(n, a_1, \ldots a_k)$ belongs to the relation p for $P_q(I)$ iff the tuple $(a_1, \ldots, a_k)$ is an answer of the subquery rooted at p evaluated over the tree rooted at the node n of I. Intuitively, the rule associated to the relation p checks if there exists a tuple u belonging to the relation p', for each child p' of p and that these tuples satisfy the join constraints over the values and the relation constraints (children and descendants) over the nodes.

We now sketch the construction of a program $\widehat{P}_q$ that computes the generalized tuples for satisfiability. It has one relation $\widehat{p}$ for each node p in q with the same arity as p and a relation $\widehat{q}$. Relation $\widehat{p}$ defines the satisfiability for the subquery rooted at p. Program $\widehat{P}_q$ is obtained as following :

1. For each rule r of P_q, the program $\widehat{P}_q$ contains the rule obtained from r by replacing each relation p by $\widehat{p}$.

2. For each node p of q that is not the root, $\widehat{P}_q$ also contains is a rule that introduces a generalized tuple $(n, x_1, ..., x_k, \varnothing)$ in $\widehat{p}$:

$$\widehat{p}(n, x_1, ..., x_k) \leftarrow function(n)$$

   where n is a variable denoting a function node and $x_1, x_2, ..., x_k$ are the other variables for the variables occurring in $\widehat{p}$.

Observe that the second kind of rules may introduce unconstrained variables. This comes from the fact that a call may a-priori bring data matching any pattern. Note also that equality constraints in the generalized tuples are introduced by the joins.

The program $\widehat{P}_q$ for the query q of Figure 2.5 is given in Algorithm 2.1. The nodes of the query are numbered using preorder traversal. The node p_1 denotes the root of the query. The six first rules are derived from P_q. The last four rules are added to compute the satisfiable tuples in CQL.

Algorithm 2.1 Program $\widehat{P}_q$ for Query q of Figure 2.5

begin
1 : $\widehat{q}(x, y) \leftarrow \widehat{p_1}(n, x, y)$
2 : $\widehat{p_1}(n, x, z) \leftarrow root(n), label(a, n), child(n, n'),$
$child(n, n''), \widehat{p_2}(n', x), \widehat{p_4}(n'', y, z), x = y$
3 : $\widehat{p_2}(n, x) \leftarrow child(n, n'), label(b, n), \widehat{p_3}(n', x)$
4 : $\widehat{p_3}(n, x) \leftarrow label(x, n)$
5 : $\widehat{p_4}(n, x, y) \leftarrow label(x, n), child(n, n'), \widehat{p_5}(n', y)$
6 : $\widehat{p_5}(n, y) \leftarrow label(y, n)$
7 : $\widehat{p_2}(n, x) \leftarrow function(n)$
8 : $\widehat{p_3}(n, x) \leftarrow function(n)$
9 : $\widehat{p_4}(n, x, y) \leftarrow function(n)$
10 : $\widehat{p_5}(n, x) \leftarrow function(n)$
end

In [Kanellakis 95], it is shown how to evaluate datalog on generalized tuples in PTIME. Using their result, one can show that:

Theorem 2.6. *Let q be a query. Then there exists an nrec-datalog program $\widehat{P}_q$ such that for each generalized n-tuple $(u, \mathcal{C})$, (sound) if $(u, \mathcal{C}) \in \widehat{P}_q(I)$, then $(u, \mathcal{C})$ is satisfiable for (I, q) and (complete) if $(u, \mathcal{C})$ is satisfiable for (I, q), then there exists $(u', \mathcal{C}')$ in $\widehat{P}_q(I)$, $(u, \mathcal{C}) \sqsubseteq (u', \mathcal{C}')$. Given I, one can compute $\widehat{P}_q(I)$ in* PTIME *in the size of the document.*

The proof is given in Appendix A.2.

Observe that the set of tuples returned by the program $\widehat{P}_q$ may be exponential in the size of q. To analyze more precisely the complexity, we turn to Boolean queries. It is interesting to note that a generalized tuple is satisfiable for some (I, q) if the Boolean query $q(\theta(u))$ is satisfiable for I for some θ that maps each variable in u to a distinct new constant not occurring in I or q. We now consider the complexity of deciding whether a Boolean query is satisfiable for some document.

Theorem 2.7. *Given I and a Boolean query q, one can decide whether q is satisfiable for I in* PTIME *in the size of I. The problem is* NP-COMPLETE *in the size of q (or the size of I and q).*

A detailed proof is given in Appendix A.2. We provide next a sketch of that proof.

Proof. (sketch) The data complexity follows from Theorem 2.6. NP-hardness is by reduction of the evaluation problem that is known to be NP-COMPLETE. See Theorem 7.3 of [Gottlob 02]. We now prove that the satisfiability problem is in NP. Consider an instance (I, q) of the problem. To show that $I \models \diamond q$, it suffices to exhibit a sequence ω of insertions and a valuation ν of q in $\omega(I)$. First, observe that if such a sequence exists, there is one with a number of insertions bounded by $|q|$ and the size of inserted trees also bounded by $|q|$. Furthermore, observe that we need only to consider a polynomial number of labels (we have to guess values for variables). Then we have to check (in polynomial time) that the given *candidate valuation* is successful. So, a polynomial number of guesses (to guess a valuation) followed by a polynomial computation (to check the valuation) suffice. This shows that the problem is in NP. □

This result can be extended to Axlog systems.

Axlog system and recursion One can consider an Axlog system. Recall that in each document, queries produce streams of answers. These streams are used as input streams to other documents of the system

The datalog computation can be generalized to this setting. Observe that the program may now be recursive. The following theorem considers the data and combined complexities for such systems.

Theorem 2.8. *Let $\mathcal{S}$ and (d, q) be an Axlog system with empty queues and an Axlog widget belonging to this system. Then there exists a datalog program $\tilde{P}_{\mathcal{S},q}$ such that for each generalized n-tuple $(u, \mathcal{C})$, (sound) if $(u, \mathcal{C}) \in \tilde{P}_{\mathcal{S},q}(\mathcal{S})$, then $(u, \mathcal{C})$ is satisfiable for (d, q) and (complete) if $(u, \mathcal{C})$ is satisfiable for (d, q), then there exists $(u', \mathcal{C}')$ in $\tilde{P}_{\mathcal{S},q}(\mathcal{S})$, $(u, \mathcal{C}) \sqsubseteq (u', \mathcal{C}')$. Given $\mathcal{S}$, one can compute $\tilde{P}_{\mathcal{S},q}(\mathcal{S})$ in* PTIME *in the size of the documents of $\mathcal{S}$.*

The document satisfiability problem for Axlog-system is EXPTIME-COMPLETE *in the size of the system.*

The proof is given in Appendix A.2.

We finish this study with two remarks : the first is about a subclass of queries for which the problem is easier and the other discusses an extension.

Remark 2 (no-join queries)**.** The program complexity of the problem is NP-COMPLETE. The complexity comes from the joins. Indeed, one can check whether a Boolean no-join query q is satisfiable for a document I in $O(|q| \times |I|)$. This is based on $\widehat{P}_q$ of Theorem 2.6 and using [Gottlob 02, Miklau 04].

Remark 3 (active data)**.** We can also consider insertions of active data (i.e., data including new call Ids). For this particular problem, such feature has no real impact on the complexity of the problem.

2.4. Typed documents

A schema introduces types for the document (as in DTD [DTD]) and for the return values of calls (as in WSDL [WSDL]). We consider types for unordered unranked trees inspired by DTDs. We study the complexity of satisfiability for queries over documents with schemas specified with such types. Results on query satisfaction (in the classical sense) by static documents constrained by DTDs can be found, e.g, in [Benedikt 08, Björklund 08, David 08, Figueira 09, Miklau 04].

The proofs are only sketched here. Detailed proofs may be found in Appendix A.3.

2.4.1. Schema definition

DTDs have been defined for unranked ordered trees. We adapt them to our context of unranked *unordered* trees. Recall that we assumed that a call Id occurs at most once in the document. On the other hand, we will accept that a document contains several calls, e.g. $?f_1, ?f_2$ to the same function, say w. (For instance, $?f_1$ may correspond to the call $w(0)$ and $f_2?$ to the call $w(1)$.) We assume the existence of an infinite set $\mathcal{F}$ of function names. The types we use are based on cardinality constraints on children of nodes. For example, the following "unordered-DTD" Δ_1 defines all trees that have a root labeled a with one b-child, any number of c-children (the nodes labeled b or c are leaves), and at least one call to some function w that returns (in each of it messages) only one node labeled c:

d	$root : a$
	$a \rightarrow \lvert b\rvert = 1 \;\&\; \lvert c\rvert \geqslant 0 \;\&\; \lvert w\rvert \geqslant 1$
	$b \rightarrow$
	$c \rightarrow$
call w	$root : c$
	$c \rightarrow$

Formally, a *cardinality constraint* over some set E is a Boolean combination of expressions of the form $|e| \geqslant k$, for some $e \in E$ and integer k. A multiset M of E is a function from E to $\mathbb{N}$. A multiset M satisfies a cardinality constraint, denoted $M \vdash C$, if by replacing each $|e|$ by $M(e)$, the cardinality constraint is true.

We use a particular symbol, namely *dom*, to represent the set of *data values*, i.e., the elements of $\mathcal{L}$ except those labels occurring in the type definition.

Definition 2.9. An *unordered-DTD* is an expression (τ, Σ, F, r) (denoted τ when the other symbols are understood) where Σ is finite set of labels, F a finite set of function names, $r \in \Sigma$ is the root label and τ maps each label in Σ into a cardinality constraint over $\Sigma \cup F \cup \{dom\}$. The *satisfaction* of an unordered-DTD (τ, Σ, F, r) by a tree (t, λ) with labels in $\mathcal{L} \cup \mathcal{F}$, denoted $t \models \tau$, is defined as follows: (i) the root must be r; (ii) the nodes with labels in $\mathcal{L} - \Sigma$ or in F are leaves; (iii) the children of each node with label in L must satisfy the corresponding cardinality constraints. The function τ is called the *children constraints* of (τ, Σ, F, r). A tree satisfies the children constraints τ iff it satisfies the previous properties (ii) and (iii).

Based on these types, we define schemas. (We consider single document schemas but this can be easily generalized.)

Definition 2.10. A(n *Axlog) schema* Δ is an expression (d, F, ζ) where d is the name of the document, F is a finite set of function names and ζ is a function associating to the document name an unordered-DTD and to each F in F another unordered-DTD. Intuitively, the unordered-DTD

associated to the document imposes the shape of the document and each unordered-DTDs associated of a function name imposes the shape of data brought by the calls of this function.

An *instance I* of a schema $\Delta = (d, F, \zeta)$ is an expression (t, λ, ν) such that the pair (t, λ) is an active document, ν is a function that maps each call Id to a function name and is the identity on $\mathcal{L}$ and the tree $(t, \lambda \circ \nu)$ satisfies $\zeta(d)$.

Our notion of active documents constrained by an Axlog schema is closely related to the notion of incomplete trees of [Abiteboul 06].

Reasoning about DTDs and reduced trees is not obvious. The following DTD τ describes the trees that have a root labeled r and at least two leaves labeled b.

$$\begin{array}{rcl} \text{root} & : & r \\ r & \longrightarrow & |b| \geqslant 2 \\ b & \longrightarrow & \end{array}$$

In Figure 2.6, Tree t_1 satisfies τ but t_1 is not a reduced tree. Indeed, the reduced tree from t_1 is t_2. But t_2 has only one leaf labeled b. So, t_2 does not satisfy τ. Observe that all trees satisfying τ are not reduced and their associated reduced trees are all t_2. So, even if the set of trees satisfying τ is not empty, the set of reduced trees satisfying τ is empty.

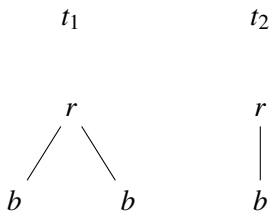


Figure 2.6.: A tree and its reduced tree

2.4.2. Satisfiability for Axlog schemas

A schema constrains the evolution of an active document. In particular, some insertions may be inconsistent if they try to transform the instance into an instance not satisfying the schema. For example, consider the Axlog schema Δ_1 previously defined. The insertion $add(?f, c)$ applied to the instance I_3 of Figure 2.4 gives an instance of Δ_1, by considering that the Id call $?f$ is a call to the function w. But the insertion $add(?f, e)$ leads to a violation of the schema constraint. We assume that an insertion that does not verify the typing constraints is simply rejected. Given a document I satisfying Δ_1, an insertion $\omega = add(?f, K)$ is *valid* if K verifies the signatures of the service corresponding to $?f$ and if $\omega(I)$ verifies Δ_1. A sequence $\omega_1; ..., \omega_k$ of insertions is considered *valid* if for each $1 \leqslant i \leqslant k$, ω_i is valid for $\omega_1; ...; \omega_{i-1}(I)$. Observe that for some set of insertions, some sequencing of the set may be valid and some invalid. The document satisfiability problem is extended to take schemas into account: a query is satisfiable over an instance of a schema (d, F, ζ) iff there is a valid sequence ω of insertions s.t. $\omega(I) \models q$.

The following theorem shows that the document satisfiability problem is still tractable in presence of unordered-DTDs with respect to data complexity.

Theorem 2.11. *Let a query q and an Axlog schema $\Delta = (d, F, \zeta)$ be fixed. Given I, the satisfiability problem for q, Δ and I an instance of Δ, is in* PTIME *in the size of I.*

Proof. (sketch) To check if a tuple u is satisfiable, we proceed as follows. First the tuple must be satisfiable in absence of typing constraints. We compute the satisfiable tuples and find all those that "cover" u. We prove that u is derived iff there exists a sequence of updates whose length is bounded by a polynomial in the size of the query. Then one has to check, for each call that is performed, that it can bring data matching the desired pattern without violating the typing constraints. These tests are hard with the respect to the query but in PTIME with the respect of the data. □

The following theorem shows that the combined complexity remains unchanged in general in the context of DTDs, but it becomes hard for no-join Boolean queries.

Theorem 2.12. *Let an Axlog schema $\Delta = (d, F, \zeta)$ be fixed. Given I and q, the satisfiability problem for q and I constrained by Δ is* NP-COMPLETE *in the size of I and q. It is already* NP-HARD *for* no-join *Boolean queries.*

Proof. (sketch) Membership in NP is proved by exhibiting a sequence of updates such that the length of the sequence is bounded by a polynomial in q and the size of each tree is bounded by a polynomial in q. NP-HARDNESS for no-join Boolean queries is proved by reduction of the satisfiability problem of a no-join Boolean query over a fixed DTD, which is known to be NP-COMPLETE. See Theorem 4.5 of [Benedikt 08]. In this proof, we use results (as Theorem 4.5 of [Benedikt 08]) and techniques for trees that are not reduced. In order to use them on our setting of reduced trees, an unordered-DTD τ is translated to an unordered-DTD τ' that "almost" describes the corresponding reduced trees. It captures the trees that can be extended to reduced trees obeying the constraint τ. □

Remark 4. When active insertions are allowed, the complexity remains the same, i.e. NP-COMPLETE in the size of the query and the document and PTIME in the size of the document.

Remark 5. The problem becomes undecidable in the case of active insertions when one considers richer typing, namely bottom-up tree automata. A complete study for such richer typing is not pursued in this thesis.

2.5. Nonmonotonicity

We consider in this section a number of nonmonotonic mechanisms, namely, deletions, end of calls, time queries and queries with negation. In this context, satisfiability is no longer monotonic; e.g. a Boolean query that is satisfiable may become unsatisfiable during the evolution of document. We first consider mechanisms so that the document is no longer inflationary. We then consider nonmonotonic queries. Details of the proofs may be found in Appendix A.4.

Noninflationary documents We consider two mechanisms that lead to a noninflationary behavior of documents: deletion and end of calls. A *deletion* is a new kind of update of the form $del(?f, q)$ where q is a tree pattern to select the nodes to delete. More precisely, the result of applying $del(?f, q)$ to a document I, denoted $del(?f, q)(I)$, is the document obtained by deleting

the siblings of node $?f$ satisfying q, as well as their descendants. (In practice, a deletion often uses identifiers to specify the subtrees to be deleted.) We also introduce the possibility that a call terminates. Formally, we also consider messages of the form $eos(?f)$, for end of update stream $?f$. When such a message arrives, the function call node is deleted. Observe that all the operations we consider, insertions, deletions and eos, are in some sense local.

Because of deletions, satisfaction is no longer monotonic. Because of deletions and eos, the document is no longer inflationary and therefore, satisfiability also is no longer monotonic. One can prove that deletions do not increase the complexity of the satisfiability problem in the simple model. Theorems of Section 2.3 remain valid in presence of deletions. But, deletions and schema constraints* together make the satisfiability problem more difficult. More precisely, Theorems 2.11 and 2.12 become :

Theorem 2.13. *Let a query q and an Axlog schema $\Delta = (d, W, \zeta)$ be fixed. The satisfiability problem for q and an instance of Δ I in presence of additions and deletions is* co-NP-HARD *in the size of I.*

Let an Axlog schema $\Delta = (d, W, \zeta)$ be fixed. The satisfiability problem for q and I constrained by Δ in presence of additions and deletions is Σ_2^P in the size of q and I.

Remark, we have considered only continuous functions. In practice, some functions are one shot, i.e. they send a unique forest as an answer, and terminate. Such only one shot Web services do not change much our setting. On the other hand, one may want to impose that the one-shot answer is a single tree. The satisfiability problem for no-join Boolean queries becomes NP-HARD when such answers are considered.

Nonmonotonic queries We next consider two kinds of nonmonotonic queries: time queries and queries with negation.

Inequations over data values and in particular over time (time constraints) in many real life examples. For instance, one may want to detect large-amount mail orders that took more than 2 days to be processed. We next sketch an extension of the model and the query language to support time-based queries relying on systems of inequations. We assume that the definition of an instance I includes a time function ψ from $nodes(I)$ to $\mathbf{Q}$. In general, it would be interesting to also consider data values from $\mathbf{Q}$ and inequations involving data values. To simplify, this is not done here. We impose that in an instance, the time of a node is larger or equal to that of its parent. Furthermore, when applying an update $add(?f, K)$ to an instance I, we impose that (i) the time of each node in K is larger than the time of each node in I; and (ii) the times of all nodes in K are identical. Condition (i) is compulsory to be able to reason about time. Condition (ii) can be relaxed but is used here to simplify.

Definition 2.14. A *time-based query* is a pair (q, C) where q is a query and C is a system of linear inequations over the nodes of q. A *valuation* ν of a query (q, C) in an instance $I = (t, \lambda, \psi)$ is a valuation of q in (t, λ) such that the system of inequations obtained by replacing each node n in C by $\psi(\nu(n))$, is satisfied.

An example of time-based document I and one of time-based query (q, C) are given in Figure 2.7. In the graphical representation of documents, we append the time to the label, as in "$a : 2$" for label a and time 2. We use a similar notation in queries.

Satisfiability is defined based on this extended notion of valuation in the obvious way. Satisfiability can also be computed in CQL, but the construction is more intricate than previously. We

*The schema may specify the nature of updates, inserts or deletes of the functions occurring in it. Details omitted.

now have to carry along each generalized fact, constraints on its variables. All this can be captured by datalog with constraints [Kanellakis 95]. A difficulty is that we don't have the time value of the future data to come. It is important to take into account the fact that this data will have a time larger than the largest time value in the instance (that we can view as the current time). Note that, as a consequence, satisfiability is no longer monotonic. Indeed, the arrival of some data that is seemingly unrelated to the query may turn some query from satisfiable to unsatisfiable simply by updating the current time. To illustrate, consider I and (q, C) in Figure 2.7. The query is satisfiable for this document. It suffices that $?f$ returns some node labeled d with time say 4. Now suppose that instead, it is $?g$ (seemingly unrelated) that returns some new node with time 10. This is imposing a new constraint on the time of data that will be received later, that makes the query unsatisfiable for the new instance.

Theorem 2.15. *Satisfiability for time-based documents and queries can be computed by a datalog program with linear inequations as constraints. Thus it can still be tested in* PTIME *in the size of the instance. It is* NP-COMPLETE *in the document and query size.*

Proof. (sketch) To prove the PTIME bound, we adapt the program $\widehat{P}$. 3Sat can be reduced to query satisfaction that itself can be reduced to query satisfiability. This shows NP-HARDNESS. □

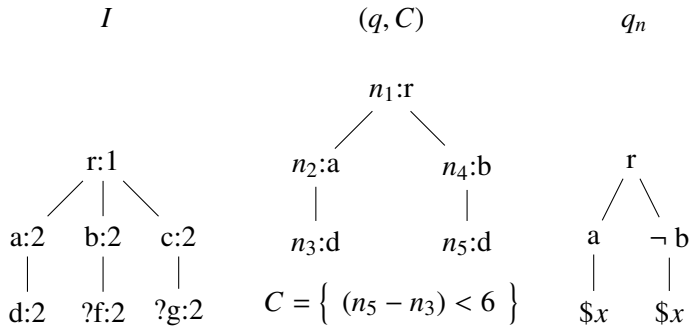


Figure 2.7.: Nonmonotonicity: Time and negation

To complete the extensions, we consider negation in queries. A tree pattern query with negation is a tree-pattern with an additional unary relation *Neg* over the nodes of the tree-pattern. The meaning of a negation is to state that "there is no subtree matching the pattern". For instance, the Boolean query q_n in Figure 2.7 states that the r-root has an a-child with a child with some label $\$x$, such that there is no b-child of the root that also has a child labeled $\$x$. In general several negations may be found on a down path from the root. The meaning of the quantification is as follows. Variables not occurring in the output are existentially quantified in the least ancestor of the nodes where they occur. Finally, we also impose that for each variable occurring in the output, at least one of nodes labeled by the variable has all its ancestor non negated.

Theorem 2.16. *The satisfiability problem is undecidable for queries with negation.*

Proof. (sketch) The proof is by reduction of the implication problem of functional and inclusion dependencies [Chandra 85]. This proof is in the spirit of that of Theorem 4.5 of [Abiteboul 06]. Our setting is more general. □

We observe that in absence of joins, the satisfiability problem is decidable for queries with negation. In the Boolean case, it has PTIME data complexity and EXPTIME-COMPLETE combined complexity. It is interesting to obtain restrictions that make satisfiability decidable even in presence of negation and joins. Such restrictions are considered, for instance, in [Abiteboul 09].

2.6. Relevance

We are interested in this section in the possible contributions of call Ids to the result of a query. This problem is particularly useful for optimization. For instance, if we know that a call does not affect the view we want to maintain, we can discard it. We introduce a notion of relevance that captures the intuition that one particular call brings useful information for some particular query we are interested in. We consider here the core model of Section 5.4 with insertions only and without time. A study of relevance with nonmonotonic features such as deletions is more complicated, and is left for future research.

After some brief motivation, we introduce a semantic notion of relevance and consider its complexity. We then introduce a weaker notion, axlog-relevance, and discuss its completeness and complexity.

Intuitively, a call Id $?f$ is "not relevant" for I, q if ignoring the data brought by $?f$ in I does not change the result of q. Consider I and q in Figure 2.8. Note that $?h$ is not relevant for I and q because its e parent does not match either b or d. Also, $?f$ is not relevant because some sibling already provides the c. On the other hand, $?g$ is clearly relevant for I and q since it can bring a node matching d.

This notion is related to the notion of *lazy-relevance* considered in [Abiteboul 04a]. They studied the closely related problem: given a query over a document with intentional data (described as call Ids), what are the Web services that need to be called to give all the mappings from the query to the active documents. Lazy-relevance can be computed using a tree-pattern query. However, the notions of relevance we consider here are more refined. For instance, the call Id $?f$ in I of Figure 2.8 is lazy-relevant for I and q, whereas one can see that it is not useful since the only matching data it can bring is a c and we already have one. We observe in passing that the notion of relevance studied here could be used to improve the query optimization technique considered in [Abiteboul 04a].

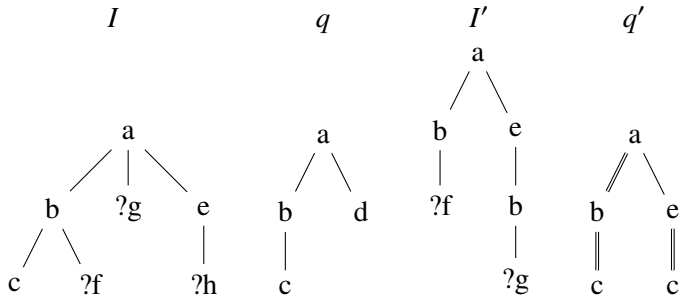


Figure 2.8.: Queries and active documents for relevance

To define relevance, we use the following auxiliary notion. Given a sequence ω of insertions and a call Id $?f$, let $\omega_{no\text{-}f}$ denote the sequence obtained from ω by removing all $?f$-insertions. Now, we have:

Definition 2.17. Let q be a query and I an active document. A call Id $?f$ is said to be *not relevant* for q and I iff for each update sequence ω and for each tuple u, $u \in q(\omega(I))$ iff $u \in q(\omega_{no\text{-}f}(I))$.

This notion of relevance of calls can be carried to data. When the data in a subtree is no longer useful (after all tuples that could be derived using it have been derived), it is not necessary to keep it. The subtree can then be garbage-collected. This will not be considered here.

The notion of relevance is somewhat more complex that it may look. Indeed, the active document I' and query q' in Figure 2.8 illustrate a subtlety. Consider the sequence $\omega = add(?f, e[c]); add(?g, c)$, that yields a document satisfying the query. A superficial analysis would lead to believe that $?f$ and $?g$ are both useful because $?f$ brings data matching the left branch of the query, and $?g$ data for the right branch. However, observe that the update $(?g, c)$ alone is enough to yield a document satisfying the query. Indeed, the update $add(?f, e[c])$ is not needed in that particular sequence and more generally, $?f$ is not relevant for I' and q' as in Figure 2.8.

Indeed, one can show that:

Theorem 2.18. *The problem of deciding, given a document and an arbitrary query, whether a call Id is not relevant is in Σ_2^P-complete in the size of the document and the query. The problem of deciding, given a document and a no-join Boolean query, whether a call Id is relevant, is* NP-COMPLETE *in the size of the query and the document.*

Proof. The Σ_2^P-hardness is shown by reducing the problem of deficiding if a tuple is a critical tuple for a conjunctive query [Miklau 07].

We prove that the relevance of a call Id is Σ_2^P. Let I be a document containing a call Id $?f$ and q a query. Then $?f$ is relevant for (I, q) if there exists an update ω, such that $\omega(I) = (t', \lambda')$, a tuple u and a instanciation θ of the variables in q such that:

(*) for each instanciation θ' from the variables in q to $\mathcal{L}$, such that for variable $\$x$ labeling a node of π, $\theta(\$x)) = \theta'(\$x)$,

(+) $\omega(I) \models \theta(q)$ and $\omega_{no-f}(I) \not\models \theta'(q)$.

First observe that it clearly suffices to consider ω of polynomial size. Also, the test (+) can be performed in PTIME. Thus the problem is in Σ_2^P.

Now consider no-join Boolean queries. NP-HARDness is by reduction of 3-SAT. For membership in NP, let $?f$ be a call Id in a document that is relevant. Then there exists a sequence of updates that demonstrates that $?f$ is relevant. One can show that there also exists a "small" sequence of updates that demonstrates it. So, to compute relevance in NP, it suffices to guess a "small" sequence of updates and test (*). The test (*) can be performed in PTIME for no-join Boolean queries. □

The previous result shows the high expression complexity of the problem. We next show that for any fixed query, relevance can be computed in PTIME in the size of the document. To do that, we explore in more details the possible scenarios for obtaining answers to the query, where a scenario is essentially assigning the different roles, i.e. the subqueries to match, to existing data or Id calls occurring in the document. More precisely, we reconsider satisfiability. We extend the generalized tuples used to describe satisfiability by including some "provenance" information. Generally, provenance is used to capture where data *came from*. Here, we are concerned with where data *might come from*. We use extended generalized tuples that are called *scenarios*. A scenario is an expression of the form $\widehat{p}(u_1, ..., u_n, \mathcal{C}, \mathcal{P})$, where each u_i is some constant or a variable and $u_1, \cdots, u_n$ is a tuple over the variables of subquery p that appear in the result or appear at least twice in q (joins), $\mathcal{C}$ is the set of constraints, and $\mathcal{P}$ is the provenance information defined as follows.

Let $\widehat{p}(u_1, ..., u_n, \mathcal{C}, \mathcal{P})$ be a generalized scenario derived for some query node $\widehat{p}$. The provenance $\mathcal{P}$ is a tuple that specifies how the derivation of corresponding facts depends on the arrival (in

certain streams) of data satisfying certain patterns. More precisely, the provenance is an m tuple, where m is the number of nodes in the subquery rooted at $\widehat{p}$. The k-th component of $\mathcal{P}$ corresponds to the k-th node of the subquery, in some fixed ordering of these nodes, say preorder traversal. Its value is $\star$ if some data is already present in the document and matches the corresponding query node. It is $n_{?f}$ for some call Id $?f$ if this specific call Id can bring data matching it. It is $\bullet$ otherwise, with the meaning that the data comes from a match in an ancestor node.

We modify the datalog program that computes satisfiability so that it also computes provenance information. For the query q and the document I of Figure 2.8, three scenarios are derived (by considering the nodes of the query with the prefix order):

- $((), , \star, \star, \star, ?g)$,
- $((), , \star, \star, ?f, ?g)$,
- $((), , \star, ?g, \bullet, ?g)$

where the 4 entries of each tuple correspond to provenance for the 4 query nodes in preorder traversal of the query tree. (The query is Boolean, so there is no data to return). Note that each tuple corresponds to a scenario for the possible future derivation of the same fact $q()$. By observing these tuples, one may be led to believe that $?f$ or $?g$ may bring useful data. But since we already obtained the subgoal $a/b/c$, it turns out that this is not the case and only $?g$ is relevant. So, we have to check each scenario if it is "useful", i.e., if it possibly brings new results.

An instance ω of a given scenario is a sequence of updates, where each update $add(?f, K)$ in ω corresponds to some occurrence of $?f$ in the provenance for a position corresponding to a subquery rooted at p if the edge between p and its parent is a *parent* edge (single line). Furthermore K satisfies the query $\theta([p])$ ($[p]$ is here the subquery rooted at p) where θ assigns to the result and join variables the values specified by this scenario. (If the edge between p and its parent is a descendant edge, some subtree of K must satisfy it, i.e., double line.)

The PTIME algorithm is rather intricate. Its crux is to check (for some I and $?f$) for each scenario where $?f$ occurs, whether there exists a tuple that would be derived in this scenario, and would not have been derived if $?f$ were removed from the scenario. This leads to:

Theorem 2.19. *Let q be a fixed query. The problem of deciding, given a document and a call Id $?f$ in it whether $?f$ is relevant for I, q, is in* PTIME *in the size of I.*

Proof. Let I be a document and $?f$ a call Id in it. We can compute in PTIME all the possible scenarios for (I, q) (i.e., the satisfiable tuples with their provenance). Consider one particular scenario $(u, \mathcal{C}, \mathcal{P})$ including $?f$. (There are polynomially many such scenarios.) Suppose (to simplify the presentation and without loss of generality) that in this scenario $?f$ is used only once. Suppose it is matched to the subquery p. Now consider the query q' obtained from q by pruning out the p subtree. The scenario gives us a scenario for q' that we call the *no-f* scenario of $(u, \mathcal{C}, \mathcal{P})$. To check that $?f$ is relevant for (I, q), it is necessary and sufficient to find a scenario $(u, \mathcal{C}, \mathcal{P})$ and a complete tuple u such that there exists an instance ω of the scenario such that:

(a) the instance ω transforms I into I' with $u \in q(I')$,
(b) the instance ω without $?f$ (which is an instance of the no-f scenario of $(u, \mathcal{C}, \mathcal{P})$) transforms I into I'' with $u \notin q(I'')$.

First observe that it is possible to restrict our attention to a polynomial number of u tuples. Let u be such a tuple. It is rather easy to test (a). The test of (b) is trickier. Consider the query $\tilde{q}$ obtained by

transforming I as follows: the $?f$ call and the $?g$ calls not occurring in the scenario are removed, a $?g$ call occurring in the scenario is replaced by the subquery it is supposed to provide according to this scenario. This query (almost) tests whether a document comes from this particular scenario omitting $?f$. Indeed, one can show that $?f$ is relevant iff there exists an active document J such that $J \models (\tilde{q} \wedge \neg q(u))$, i.e. $\tilde{q} \not\subseteq q(u)$. Intuitively, from such a J, one can construct an update that demonstrates that $?f$ is relevant. This query containment can be tested in PTIME in the size of I. To do that, we eliminate the joins by considering all valuations of the join variables. This results in replacing the containment test by many simpler containment tests. This kind of containment test can be done in PTIME in the size of I by using Theorem 1 of [Miklau 04] □

Observe that this technique leads to a lot of computation for each satisfiable tuple. One can avoid testing many of them using a notion of "dominance". For instance, in the previous example, t_1 dominates t_2 and t_3, so we find immediately that it is the only scenario to consider and we derive that $?f$ is not relevant. This suggest a necessary notion of relevance that we study to conclude this section. It is the one that is used in our system [4].

A *renaming* of a generalized tuple t is a tuple t' obtained by renaming (using a bijection) the variables of t. Let

$$t_1 = (u_1, ..., u_m, \mathcal{C}_1, \mathcal{P}_1),\ t_2 = (u'_1, ..., u'_m, \mathcal{C}_2, \mathcal{P}_2)$$

be two tuples with provenance. (We assume without loss of generality that they have the same "data" part consisting of distinct variables). We say that t_1 is *dominated* by t_2, denoted $t_1 \prec t_2$ if (a) $(u_1, \ldots, u_m, \mathcal{C}_1) \sqsubseteq (u'_1, \ldots, u'_m, \mathcal{C}_2)$ and (b) for each $p \in nodes(q)$, either $\mathcal{P}_2(p) = \star$ or $\mathcal{P}_1(p) = \mathcal{P}_2(p)$ and there exists at least one p such that $\mathcal{P}_2(p) = \star$ and $\mathcal{P}_1(p) \neq \star$. The intuition is that any relevant data needed by the dominating tuple to lead to satisfied tuples is also needed by the dominated one. Thus, the dominated tuples are useless because they lead to the same satisfied tuples.

We refine the set of candidates by eliminating the dominated tuples. In the previous example, t_2 and t_3 are eliminated. This leads to the notion of *axlog-relevance*. Let q, I and $?f$ be a query, an active document and a call Id of I. Then $?f$ is *axlog-relevant* for q if there exists a not dominated tuple $p(u, \mathcal{C}, \mathcal{P})$ (i) that may derive new results and (ii) where $?f$ appears. In Figure 2.8, the first tuple gives a new result so only $?g$ is axlog-relevant.

The different notions of relevance are related in the following way. Relevance is more refined than axlog-relevance that is more refined than lazy-relevance. In particular, in Figure 2.8 for I and q, the Id call $?f$ is lazy-relevant but not axlog-relevant (neither relevant). Also, in Figure 2.8, for I' and q', the call Id $?f$ is axlog-relevant but not relevant.

Relevance and axlog-relevance both have PTIME complexity in the size of the active document. Axlog-relevance is more tractable in practice since the polynomial has a much smaller coefficient.

Chapter 3.

Axlog

3.1. Introduction

The work on Axlog was motivated by the development of the P2PMonitor system, a system for monitoring P2P applications, introduced in [Abiteboul 07] and developed in [Marinoiu 09]. Based on our maintenance optimization algorithm, an Axlog engine has been implemented [4]. It was demonstrated in [9] using the Dell supply chain application, together with the P2PMonitor system.

The main issue for Axlog widgets is the efficient computation of output streams, i.e. a view maintenance problem. In this chapter, we outline an algorithm for incrementally computing these output streams presented in [4] and in [Marinoiu 09]. The algorithm exploits known datalog optimization techniques such as Differential [Blakeley 86b] and MagicSet [Beeri 91] mixed with the notions of satisfiability and relevance presented in the previous chapter. Satisfiability allows stating whether some (incomplete) fact has a chance to hold in the future. We see that, with this new notion of satisfiability, our algorithm is more optimistic (aggressive) than MagicSet. Based on relevance, we show how to filter data *before* it enters the datalog program (to save on processing) and possibly at the source of the stream (to save on communication).

The chapter is organized as follows. In Section 3.2, we introduce the Axlog widget environment and briefly illustrate how Axlog widgets can be used. In Section 3.3, we outline the optimized algorithm for the management of Axlog widgets for the core model and explain how the notion of satisfiability and relevance introduced in the previous chapter are employed.

3.2. Axlog at Work

Axlog Widget

An Axlog widget is mainly a complex stream processor, that is defined by one (Active)XML document and one or several queries. The widget receives update streams and generates output streams. The content of each output stream is the new trees produced by the query of the Axlog widget, see Section 5.4. An implementation of Axlog widgets is supported by P2PMonitor, a system for managing streams. The streams in P2PMonitor are implemented by a Pub/Sub mechanism based on Web Services. More precisely, a stream is exposed as a Web service to which Axlog widgets may subscribe. (A list of subscribers is maintained for each channel by its owner). When subscribing to a channel, an Axlog widget specifies a URI, i.e. the address of a Web Service that is called by the channel provider every time a new update becomes available for that channel. In P2PMonitor, a user can define a stream query by defining an Axlog widget that may subscribe to other streams defined by Axlog widgets.

Application

We have illustrated the use of Axlog widgets by considering the Dell supply chain application [Kapuscinski 04]. This distributed application represents the computer manufacturing platform of the Dell company. It involves customers, Web stores, plants for computer manufacturing, banks, suppliers, shipping companies and warehouses for the parts used by the plants. An order issued by some customer enters the system via the Web store. After payment through a bank, the order arrives in a plant that obtains the relevant parts from a warehouse and assembles the product. Suppliers have to permanently supply this warehouse to avoid delays in obtaining the parts. Finally, the product is shipped.

In such an environment, the use of Axlog widgets facilitates supporting tasks that are typically very complex because of the distribution. For instance, they turn to be very useful to gather information from the entire system. A set of Axlog widgets may be used to help users monitor the processing of their orders. For instance, the monitoring of orders of INRIA that have been shipped can be supported using the Axlog widget of Figure 3.1. The active document of Figure 3.1 uses streams to the arrivals of new orders in the different parts of the Dell supply chain application. The query assumes that the complete data of an order are stored at the Web store level. In order to know that an order has been passed by INRIA, the query has to look at data sent from the Web store. The system supporting the Axlog widget is able to select the different sources relevant for the query. In the example, the system only needs to subscribe to the streams *orders_webstore* and *orders_shipping*.

3.3. The View Maintenance Algorithm

Our algorithm to produce the output stream is based on view maintenance techniques. The query of the Axlog widget is maintained over the active document. The updates to this view are published on an output stream. In this section, we outline how to optimize this maintenance. This is achieved by combining a wide array of existing techniques on datalog-based query processing on trees, datalog optimization and stream filtering, and introducing novel features that are more specific to active documents.

We focus here on the core model of the axlog widgets.

Datalog, Differential and MagicSet

First, a tree pattern query of an Axlog widget is translated into a datalog program in the style of [Gottlob 02]. To use datalog, we represent the document (a tree) using relations, see Section 2.3. To optimize the maintenance of this program, we use two known techniques for datalog, namely Differential for incremental computations [Blakeley 86b] and MagicSet [Beeri 91, Vieille 89] for query optimization.

Indeed, we want to avoid deriving irrelevant facts. To do that, we use the MagicSet technique, that rewrites the datalog program (given the view query) into one that derives only facts that are "relevant" for the view. Also, we want to avoid recomputing the view when new updates arrive. In this purpose, we use the Differential technique for the incremental maintenance of datalog views. This technique prevents us from unnecessarily repeating the same datalog derivations when a stream brings new data. The combination of these various techniques is already the source of important savings.

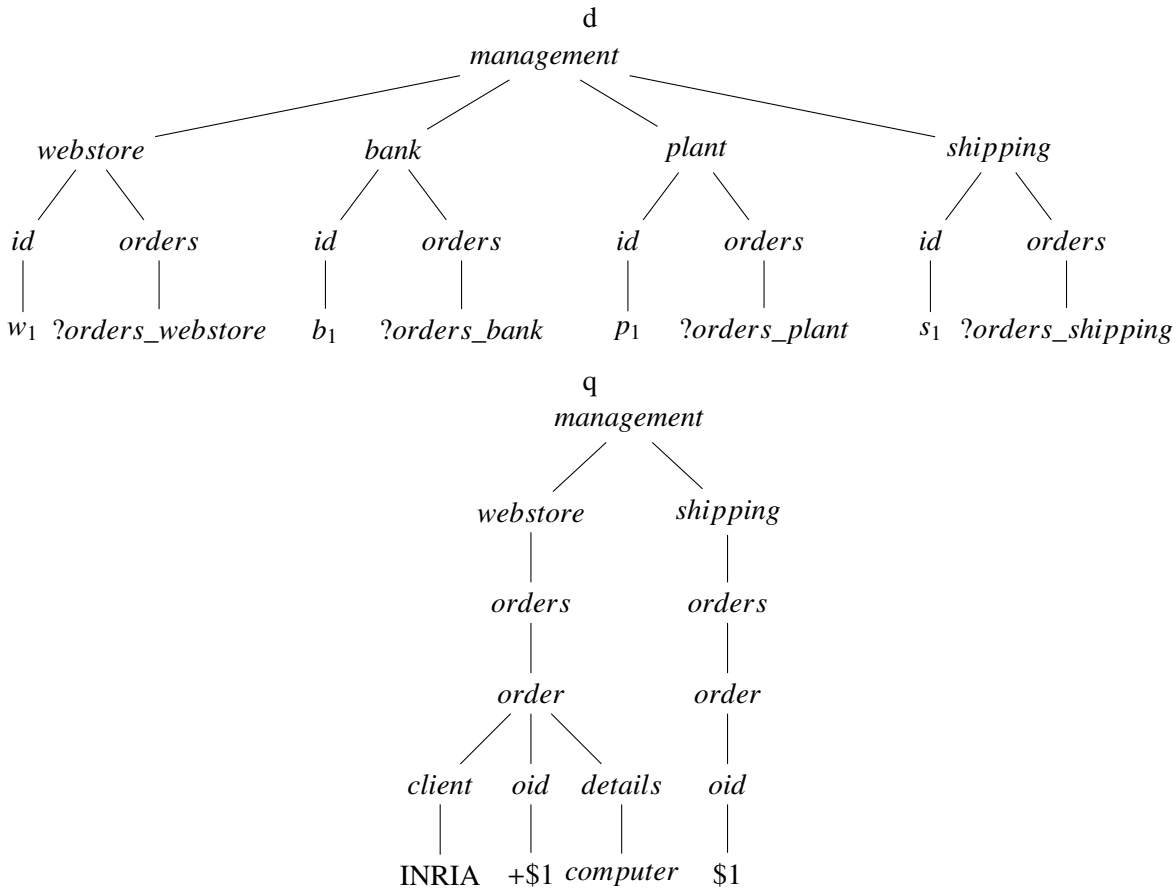


Figure 3.1.: An Axlog widget

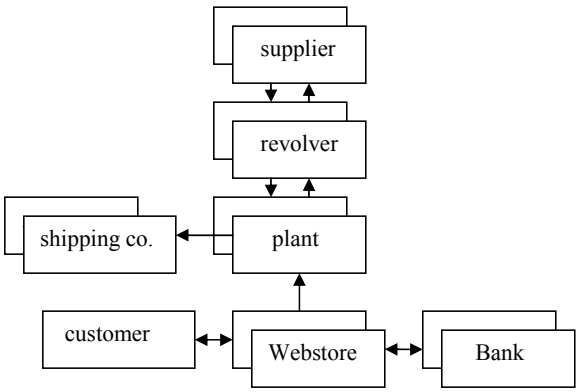


Figure 3.2.: The Dell supply chain

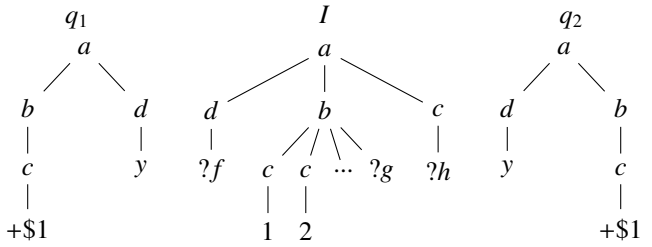


Figure 3.3.: Beyond MagicSet

MagicSet and Differential do not change the complexity of the evaluation of a query that is EXPTIME for the combined complexity in practical and PTIME for the data complexity. We next mention new optimizations based on satisfiability and relevance. These optimizations doe not change the complexity of the evaluation. In practice, the new algorithm brings serious improvements as shown in [6] and in [Marinoiu 09].

A More Optimistic Strategy using Satisfiability

The crux of MagicSet is to "focus on relevant data" (for a given query). However, some data may seem irrelevant for now but may be relevant assuming some data is received in the future. The computation of these facts is blocked with MagicSet.

To illustrate it, consider Figure 3.3. The notion of relevance as used in MagicSet heavily depends on the ordering of the query branches (e.g. left is considered before right here). When evaluating q_1, the y branch is tested before the other branch. Suppose the data consists of a large collection of subtrees having roots labeled c (brought by $?g$). Observe that until $?f$ produces a node labeled y (for yes), the c subtrees do not produce any answer. If we evaluate the query q_2 with MagicSet, no tuple is produced until the y is received. In particular, the b subtrees are not even tested. Then, when y arrives, we have to perform a lot of computation and our response time will be worse for q_2 than for q_1.

To avoid this side effect, we relax the MagicSet technique by allowing our program to derive facts that are only satisfiable. In the example, since we know that the f function call *may* return a node labeled y, it seems more appropriate to be optimistic and start testing the b subtrees in advance. If we allow to derive the facts for computing satisfiability, we more optimistically test the b-subtrees.

Filtering the Streams

First, observe that some function calls are irrelevant for the query and may be unsubscribed by the system. Moreover, observe that the computation of scenarios, see Section 2.6, with provenance information tells us more than just relevance. For a relevant call $?f$, we also find precisely for what it is relevant, i.e., the list of subqueries for which it can bring relevant data. Based on that, we can filter the stream of data brought by $?f$ to let only relevant data enter the document. This presents the advantage of reducing processing (fewer data enters the datalog program) and also communication (if the filtering is performed at a remote source).

To illustrate the use of filters, consider the query q_2 and the document I in Figure 3.3, where the p_is represent node identifiers of the query. Observe that before we can derive any result, the

tree-pattern rooted at the node labeled c has to be matched to data returned by $?g$.

However, the system may perform a lot of filtering tasks for the same stream. A first optimization is to prune some unneeded scenarios by using the notion of axlog-relevance presented in Section 2.6. Moreover, in the implementation, we use YFilter [Diao 02], one of the NFA-based solutions, because it scales very well with the number of queries on a stream. Moreover, to avoid installing too many filters on one stream, we use the notion of axlog-relevance to prune some unnecessary scenarios.

Part II.

Data-centric Workflow Applications

Overview of Part II

The evolution of shared data is at the center of many human activities consisting of tasks whose sequencing is governed by a workflow. Workflow models have traditionally been operation-centric, ignoring almost completely the data aspects. Recently, there has 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.

Towards a data-centric workflow approach, we first introduce the core of the *AXML Artifact model*, called BAXML, to capture data and workflow management activities in distributed settings. The model is built on active documents. One of the most discussed aspects of modeling data-centric workflow is the specification of the workflow constraints. 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. We compare the expressiveness of several workflow specification mechanisms, including automata, temporal constraints, and pre-and-post conditions with BAXML as underlying data models. One surprising result shows the considerable power of static constraints to simulate apparently much richer workflow control mechanisms. Moreover, we argue that the model captures the essential features of business artifacts as described informally in [Nigam 03a] or discussed in [Hull 08]. To motivate this assertion, we compare the expressiveness of the BAXML model with the tuple artifact model introduced in [Deutsch 09].

In a second part, we extend the AXML Artifact model with different features: hierarchy of artifacts, access control, dynamic workflow behavior modifications and distribution of the system. Hierachy of artifacts and distribution have already been presented in [5] and [Hélouët 10]. Distribution is a challenging problem for the implementation [Marinoiu 09] and model checking [Hélouët 10]. This extension is not studied in this thesis. The other extensions are new. We illustrate these different notions by two examples coming from Business applications: an example based on a casting procedure as described in [Wikipedia] and an example based on the Dell Supply Chain [Kapuscinski 04]. The casting procedure illustrates the three first notions. The Dell Supply Chain illustrates the distribution. Finally, we present AXART a system supporting most of the AXML Artifact model with the extended features. This system focuses on human interactions managed by a central system. Thus, the distribution is not supported by AXART.

To summarize, we present a data-centric workflow language, AXML Artifacts, based on active documents. Our main contribution, published in [2], is a study of the expressiveness of different formalisms to control the sequencing of the active documents (BAXML), the core of the AXML Artifact model. We implemented a prototype of this model, demonstrated in [8]. The distribution, not supported by AXART, is briefly discussed in this thesis. It is fully developed in [5, Marinoiu 09]

This part is organized as follows. The related work is presented in Chapter 4. In Chapter 5, we present the core model of AXML Artifact and different ways to express workflow constraints. These are compared among each other and with an other model. Chapter 6 presents extensions of the model and an implementation. The proofs are presented in Appendix B.

Chapter 4.

Related Work

Workflow languages has been extensively studied in the last 15 years to answer the need to model business process activities. In this context, several languages for specifying workflows or for modeling them have been proposed following different paradigms:

- declarative ways: specify in logic terms the sequencing of tasks
- procedural ways: specify operationally the sequencing of tasks

Procedural languages have initially been more popular, but the complexity of modeled activities leads to proposals of declarative as data centric workflows. In particular, IBM has introduced the Business Artifact model, called tuple artifact model, based on describing the evolution of relational data. In the same way, the AXML artifact model is a data centric workflow model based on describing the evolution of data represented using trees. To facilitate the development of web applications, systems relying on declarative specifications have been developed in last ten years. In parallel, their properties have been formally studied in several works.

In a first section, we present briefly different languages used for describing workflows. In the second section, we present different systems implementing workflow descriptions. In a last section, we sum up the formal works about data centric workflows.

4.1. Workflow languages and models

Workflow modeling and specification has traditionally been process centric (e.g., [Georgakopoulos 95, van der Aalst 04]). This has been captured in the workflows community by flowcharts, Petri nets [van der Aalst 98, van der Aalst 02, Adam 98], and state charts [Harel 87, Mok 02]. More recently, data-centric workflows have been considered in [Wang 05], and in particular the *artifact* model of IBM [Nigam 03b]. It was subsequently studied from both practical and theoretical perspectives in [Bhattacharya 07a, Bhattacharya 05, Deutsch 09, Bojanczyk 06a, Bhattacharya 07b, Hull 00, Hull 99, Kumaran 08, Meier 10, Libkin 04, Martin 03, Zhao 09]. The Vortex framework [Hull 99, Dong 99, Hull 00] is another way to describe declarative specifications for when a services are applicable to an artifact. Moreover, the OWL-S [Martin 03, McIlraith 01] proposal describe services using input/output, pre/post-conditions. There are used to model evolving database that is closely related to the tuple artifact model. The AXML Artifact model introduced in [5], it is inspired by [Abiteboul 09] that describes a first mechanism to sequence active documents by using guards to control the activation of function calls. Different variants have been studied to obtain formal results on reachability [Abiteboul 04b, Genest 08], model checking [Abiteboul 09, Genest 10], equivalence [Abiteboul 11] decidability results . An extension of the model presented in [Abiteboul 09] to describe interfaces has been developed in [Hélouët 10].

4.2. Workflows systems

The language BPEL, [BPEL], is the main language used in practice to build applications by orchestrating web services as a procedural workflow.

The system AXART is related to platforms that help users model Web applications, such as FORWARD [Bhatia 09] and WebML [Ceri 00]. It is also related to workflow design languages such as the Vortex system [Hull 99], Business Artifacts [Hull 08] and to mashup systems such as Yahoo Pipe [Yahoo Pipe]. The first two systems help the user describe Web applications in terms of interactions. The business artifact model is used for complex applications, while mashup systems are used to perform automatic data integration. Our system focuses on collaborative and dynamic applications. In particular, the workflow can change during the life of an artifact.

4.3. Formal studies about workflows

4.3.1. Comparison of workflow languages

The comparison of such workflows systems by using the notion of bisimulation was introduced in [Milner 89, van Benthem 76]. An example can be found in [Alur 05]: they compare the recursive state machine, a formalism based on automata allowing to create new states and pushdown system based on an automaton with a stack. It is shown that these two formalisms are bisimilar. The comparison of data-centric formalisms has not been studied since the paper [2]. The comparison of tuple artifact specifications is considered in [Calvanese 09] based on notion of dominance, that focuses on the input/output pairs of a workflow. Decidability of dominance has been proven for bounded sequences of relational tuple artifacts where services are described using Presburger formulas. In the same vein, [Abiteboul 11] looks at equivalence of AXML documents modeling a distributed query plan.

4.3.2. Verification

Verification for data-centric models based on transforming relational data is considered in [Gerede 07a, Gerede 07b, Bhattacharya 07b, Deutsch 09, Fritz 09]. Other models in the same spirit include the Vortex workflow framework [Hull 99, Dong 99, Hull 00], the OWL-S proposal [McIlraith 01, Martin 03] as well as some work on semantic Web services [Narayanan 02]. The article [Deutsch 07] (building on [Spielmann 03, Spielmann , Abiteboul 00]), considers the verification of properties specified in LTL-FO, first-order logic extended with linear-time temporal logic operators, of data-centric workflows. Similar extensions have been previously used in various contexts [Emerson 90, Abiteboul 96, Spielmann 03, Deutsch 06]. An extension [Damaggio 11] including arithmetic constraints and data dependencies has recently been studied. Most of the previous works show decidability in PSPACE of satisfaction of LTL-FO formulas for restrictions of the tuple artifact model. The latter work shows decidability of satisfaction of LTL-FO formula in HYPEREXPTIME. Most of the works on model checking on AXML have been discussed Chapter 1.

The above works are relevant to the large verification community. Model checking techniques have bee recently extended to infinite-state systems (e.g., see for a survey [Burkart 01]). More precisely, recent works are focusing on data as source of infinity mixed with recursive procedure. This includes recursive procedure with arithmetic [Bouajjani 03], rewriting system with data [Bouajjani 07b, Bouajjani 07a], Petri nets with colored token [Lazić 07], automata and logics over

infinite alphabets [Bouyer 02, Bouyer 03, Neven 04, Demri 09, Bojanczyk 06b]and temporal logic manipulating data[Demri 09, Demri 08].

Apart from the work on verification of BAXML with guards mentioned above [Abiteboul 09], most other work on static analysis on XML (with data values) deals with documents that do not evolve in time, e.g., [Fan 01, Arenas 02, Alon 03]. See [Segoufin 07] for a survey on related issues.

Chapter 5.

Comparing Workflow Specification Languages: A Matter of Views

5.1. Introduction

The evolution of shared data is at the center of most human activities. The novel notion of *business artifact* [Nigam 03a] has been proposed to specify such evolution. The main idea is to capture both the flow of control (workflow) of the application but also data evolution (data cycle). Building on active documents, we propose a new artifact model, the AXML Artifact model. The sequencing of active documents are specified by different specification mechanisms based on automata, pre/post conditions, and temporal constraints.. The main goal of this chapter is to compare them.

Comparing workflow specification languages is intrinsically difficult because of the diversity of formalisms and the lack of a standard yardstick for expressiveness. In this chapter, 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

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. Artifact systems may also impose constraints by temporal formulas on the history of the run ([Hull]).

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 chapter.

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

Depending on the specific needs, a workflow view might retain information about some abstract state of the system and its evolution, about some particular events and their sequencing, about the

entire history of the system so far, or a combination of these and other aspects. Even if not made explicit, a view is often the starting point in the design of workflow specifications. This further motivates using views to bridge the gap between different specification languages. To see how this might be done, consider a workflow W specified by tasks and pre/post conditions and another workflow W' specified as a state-transition system, both 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 workflows models to an entirely different model appropriate for the comparison. We will formalize the general notion of view and introduce a form of bisimulation over views to capture the fact that one workflow simulates another.

In our formal development, we mostly use the Active XML model [Abiteboul 08a], 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 active documents that are adapted in the context of workflows. The document evolves as a result of function calls that initiate new sub-tasks, and returns of results of function calls (using some local rewritings). The functions can be internal or external, the latter modeling interaction with the environment. For example, a BAXML document is shown in Figure 5.1. 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 09], 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 workflow 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 09]. 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.

The chapter is organized as follows. We introduce the view-based framework for comparing workflow languages in Section 5.2. The BAXML model and the workflow languages are presented in Section 5.4. Their expressive power with respect to different views is compared in Section 5.5. In Section 5.6 we compare BAXML with a variant of IBM's business artifacts, and show that BAXML can simulate artifacts, but the converse is false. Proofs are relegated to Appendix B.

5.2. Views and Simulations

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

Workflow Systems and Languages

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

The workflow systems we consider include two particular events, namely *block* and ε, both in E_∞, whose role we explain briefly. First consider *block*. For uniformity, it is convenient to assume that all runs are infinite. To this end, we use the distinguished event *block* to signal that the system

has reached a terminal state that repeats forever (so once a system blocks, it remains blocked).

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

More formally:

Definition 5.1 (Workflow System)**.** A workflow system is a tuple $(N, n_0, \delta, q_0, \lambda_N, \lambda_\delta)$ where:

- (N, n_0, δ) is a tree with root n_0, nodes N, edges δ.
- all maximal paths from n_0 are infinite.
- λ_N is a function from N to Q_∞, and $\lambda_N(n_0) = q_0$.
- λ_δ is a function from δ to E_∞.
- for each $(n, n') \in \delta$, if $\lambda_\delta((n, n')) = \varepsilon$ 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, 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 chapter, we rely heavily on the notion of view in order to compare workflows languages.

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

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

*Composition is applied left-to-right.

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 89] (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 $e \in E - \{\varepsilon\}$, we define the relation $\stackrel{e}{\rightarrow}$ on nodes by $n \stackrel{e}{\rightarrow} m$ if there is a sequence of transitions from n to m, all of which are silent except for the last one, which is labeled e.

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

Definition 5.2 (w-bisimulation)**.** Let

$$WS_i = (N^i, n_0^i, \delta^i, q_0, \lambda_N^i, \lambda_\delta^i)$$

$i \in \{1, 2\}$, 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 $e \neq \varepsilon$, if $n_1 \stackrel{e}{\rightarrow} n_1'$ in WS_1 then there exists n_2' such that $n_2 \stackrel{e}{\rightarrow} n_2'$ in WS_2 and $B(n_1', n_2')$, and conversely.
- there is an infinite path of silent transitions from n_1 in WS_1 iff there is an infinite path of silent transitions from n_2 in WS_2.

We denote by $WS_1 \sim WS_2$ the fact that there exists a w-bisimulation of WS_1 and WS_2.

We note that there are well-known notions of bisimulation related to ours, such as weak-bisimulation and observation-congruence equivalence, motivated by distributed algebra [Milner 89]. 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 5.3 (Simulation)**.** Let $\mathcal{W}_1, \mathcal{W}_2$ be workflow languages and $\mathcal{V}_1, \mathcal{V}_2$ be sets of views. The language $\mathcal{W}_2$ *simulates* $\mathcal{W}_1$ *with respect to* $(\mathcal{V}_1, \mathcal{V}_2)$, denoted $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1,\mathcal{V}_2)} \mathcal{W}_2$, if for each $W_1 \in \mathcal{W}_1$ and $V_1 \in \mathcal{V}_1$ such that $V_1(W_1)$ is well-defined, there exist $W_2 \in \mathcal{W}_2$ and $V_2 \in \mathcal{V}_2$ such $V_2(W_2)$ is well-defined and $V_1(W_1) \sim V_2(W_2)$.

Remark 6. 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 5.23 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 5.4 (Composition)**.** *Let $\mathcal{W}_1$ and $\mathcal{W}_2$ be workflow languages and $\mathcal{V}_1, \mathcal{V}_2$ and $\mathcal{V}$ be sets of views. If $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1,\mathcal{V}_2)} \mathcal{W}_2$ then $\mathcal{W}_1 \hookrightarrow_{(\mathcal{V}_1\circ\mathcal{V},\mathcal{V}_2\circ\mathcal{V})} \mathcal{W}_2$.*

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

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

Lemma 5.5. *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.

5.3. 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 09], 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. BAXMLconsists of extended active documents with function calls modeling internal computations and asynchronous interactions with the environment. It is the base of AXML Artifacts.

To illustrate our definitions, we use a simplified version of the Mail Order example of [Abiteboul 09]. 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. We assume given the following disjoint infinite sets: *nodes* $\mathcal{N}$ (denoted n, m), *tags* Σ (denoted $a, b, c, \ldots$), *function names* $\mathcal{F}$, *data values* $\mathcal{D}$ (denoted $\alpha, \beta, \ldots$) *data variables* $\mathcal{V}$ (denoted $X, Y, Z, \ldots$), possibly with subscripts.

In the model, trees are active documents with some differences. In this part, we do not keep all the informations about the call too. But contrary about in the previous part, we keep only the state

of the call and which function name is called. By this way, we can constraints the activation and the return of service calls using the different constraints. For each function name f, we also use the symbols $!f$ and $?f$, called *function symbols*, and denote by $\mathcal{F}^!$ the set $\{!f \mid f \in \mathcal{F}\}$ and by $\mathcal{F}^?$ the set $\{?f \mid f \in \mathcal{F}\}$. The union of $\mathcal{F}^!$ and $\mathcal{F}^?$ forms the set $\mathcal{C}$. Intuitively, $!f$ labels a node where a call to function f can be made (possible call), and $?f$ labels a node where a call to f has been made and some result is expected (running call). After the answer of a call at node x is returned, the call may be kept or the node x may be deleted. If calls to $!f$ are kept, f is called *continuous*, otherwise it is *non-continuous*. For example, the role of the `MailOrder` function in Figure 5.1 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.

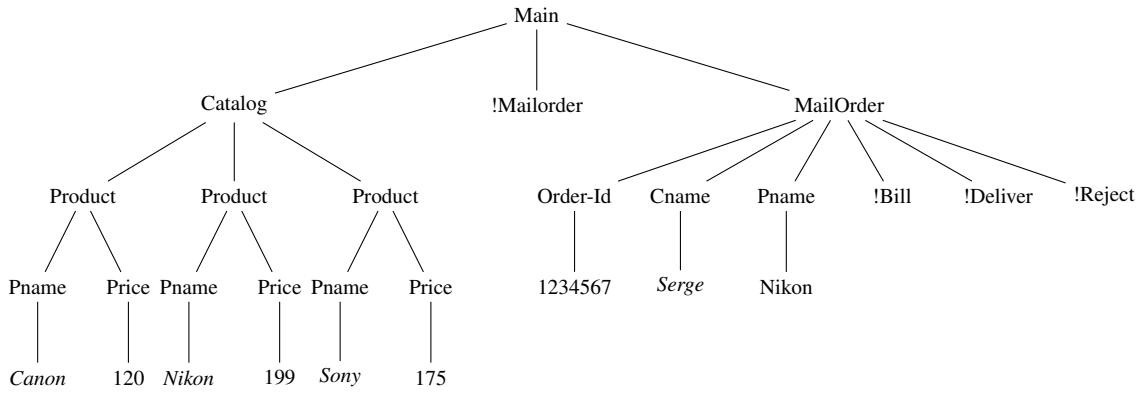


Figure 5.1.: A BAXML document.

In this chapter, we distinct values and labels. A active 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 5.1.

Moreover, we refine the notion of reduced tree to take account of the running calls $?f$. 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-pattern queries*. A *tree-pattern query* is defined as in Definition 2.2 with some differences to take account of changes in the definition of active document. 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. The nodes labeled by value variables are only mapped by nodes labeled by values of $\mathcal{V}$. 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. 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). An example is given in

Figure 5.2 (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 5.1. (Indeed, we want `Order-Id` to be a key).

We sometimes use patterns that are evaluated relative to a specified node in the tree. More precisely, a *relative* pattern is a pair $(P, self)$ where P is a pattern and *self* is a node of P. A relative pattern $(P, self)$ is evaluated on a pair (F, n) where F is a forest and n is a node of F. Such a pattern forces the node *self* in the pattern to be mapped to n. Figure 5.2 (b) provides an example of relative pattern. The pattern shown there checks that a product that has been ordered occurs in the catalog. It holds in the BAXML document of Figure 5.1 when evaluated at the unique node labeled `!Bill`.

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

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

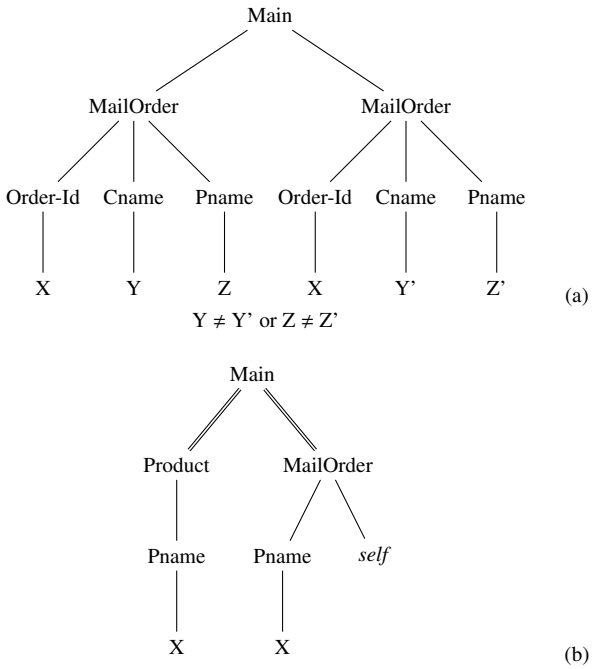


Figure 5.2.: Two patterns

Queries As previously mentioned, patterns are used in queries, as shown next. A *query* is a finite union of rules of the form $Body \rightarrow Head$, where *Body* and *Head* are patterns and *Head* contains no descendant edges and no constants, and all its variables occur in *Body*. In each tree of *Head*, all variables occur under a designated *constructor node*, specifying a form of nesting. When evaluated on a forest, the matchings of *Body* define a set of valuations of the variables. The answer for the rule is obtained by replacing, in each tree of *Head*, the subtree rooted at the constructor node with the forest obtained by instantiating the variables in the subtree with all their matchings provided by

the *Body*. The answer to the query is the union of the answers for each rule. As for patterns, we may consider queries evaluated relative to a specified node in the input tree. A *relative query* is defined like a query, except that the bodies of its rules are relative patterns (*P*, *self*). An example of relative query (with a single rule) is given in Figure 5.3. The label of the constructor node (indicated by brackets) is `Process-bill`.

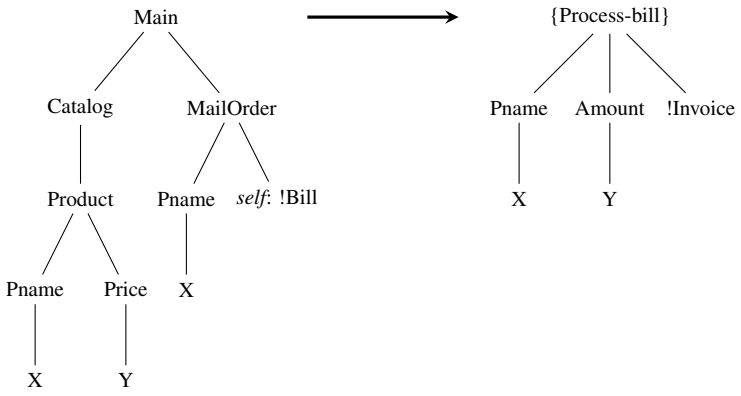


Figure 5.3.: Example of a relative query

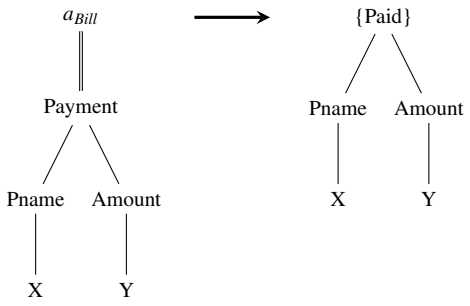
Consider the evaluation of the query of Figure 5.3 on the BAXML document of Figure 5.1 at the unique node labeled `!Bill`. There is a unique matching of the *Body* pattern and the result is the *Head* pattern of the query with *X* replaced by *Nikon* and *Y* by 199 (without brackets for `Process-bill`).

DTD Trees used by a BAXML system may be constrained using unordered DTDs and Boolean combinations of patterns. Unordered DTDs are defined as in Definition 2.9. We allow also that cardinality constraints can talk about the state of functions: they are Boolean combinations of statements of the form $|b| \geqslant k$ for $b \in \Sigma \cup \mathcal{F}^! \cup \mathcal{F}^? \cup \{dom\}$ and k a non-negative integer. Validity of trees and of forests relative to a DTD is defined in the standard way, see Definition 2.9.

Schemas and instances A BAXML *schema* s is a tuple $(\Phi_{\text{int}}, \Phi_{\text{ext}}, \Delta)$ where (i) the set Φ_{int} contains a finite set of internal function specifications, (ii) the set Φ_{ext} contains a finite set of external function specifications, and (iii) Δ provides static constraints on instances of the schema. It consists of a DTD and a Boolean combination of patterns. For instance, the negation of the pattern in Figure 5.2 (b) states that `Order-Id` uniquely determines the mail order.

We next detail Φ_{int} and Φ_{ext}. For each $f \in \mathcal{F}$, let a_f be a new distinct label in Σ. Intuitively, a_f will be the root of a subtree where a call to f is being evaluated. (This subtree may be seen as a task initiated by the function call.) The specification of a function f of Φ_{int} indicates whether f is continuous or not, provides its *argument query* (a relative query), and *return query* (a query rooted at a_f). When the argument query is evaluated, *self* binds to the node at which the call $!f$ is made. The role of the argument query is to define the argument of a call to f, which is also the initial state of the task corresponding to f.

Example 5.6. We continue with our running example. The function `Bill` used in Figure 5.1 is specified as follows. It is internal and non-continuous. The argument query is the query in Figure 5.3. Assuming that `Invoice` is an external function eventually returning `Payment` (with product and amount paid) the return query of `Bill` is:



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

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

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

We now provide more details. When a call to $!f$ is made at node n, the label of n is changed to $?f$. If f is internal, we additionally add to the graph of *eval* the pair (n, T') where T' is a tree consisting of a root a_f connected to the forest that is the result of evaluating the argument query of f on input $(\mathcal{T}, n)$. When an answer to call $?f$ at node n is received, the trees in the answer are added as siblings of n, and n is deleted (if f is non-continuous) or its label is reset to $!f$ (if f is continuous). If f is external, its answer is a forest satisfying Δ_f. If f is internal, the answer can be returned only if $eval(n)$ contains no running calls $?g$, in which case the answer consists of the result of evaluating the return query of f on $eval(n)$, after which $(n, eval(n))$ is removed from the graph of *eval*.

Figure 5.4 shows a possible next instance for the instance of Figure 5.1 after an internal call has been made to `!Bill`. Recall the specification of `Bill` from Example 5.6. As `!Bill` is an internal call, the subtree a_{Bill} contains the result of the query defining `!Bill` (see Figure 5.3). The dotted arrow indicates the function *eval*.

We will typically be interested in *runs* of such systems. An *initial* instance of schema s is an instance of s consisting of a single tree whose root is not a function call and for which there is no running call. For runs, we use a variation of the model of [Abiteboul 09]. A *prerun* of a schema s is a finite sequence $\{(I_i, e_i)\}_{0 \leqslant i \leqslant n}$, such that (i) for each i, I_i satisfies the static constraints Δ, (ii) $e_0 = init$, and (iii) for each $i > 0$, $I_{i-1} \vdash_{e_i} I_i$. A *run* is an infinite sequence $\rho = \{(I_i, e_i)\}_{i \geqslant 0}$ such that:

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

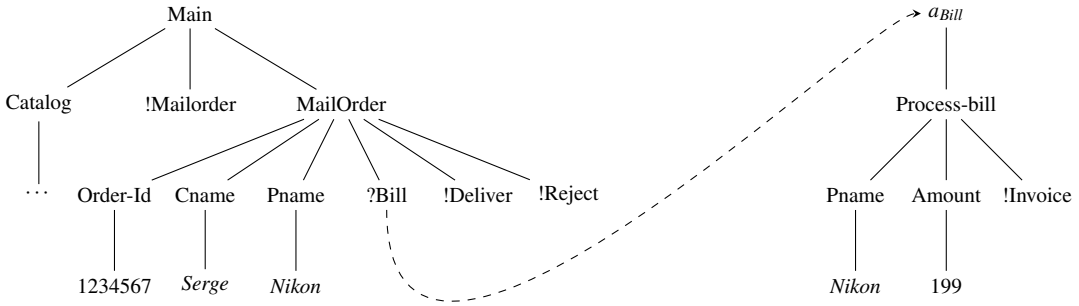


Figure 5.4.: An instance with an *eval* link

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

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

Semantics with and without aborts

We next discuss a subtle difference between the semantics adopted here and that of [Abiteboul 09]. 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 09] 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 5.7. *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† DTDs.*

The proof for arbitrary DTDs is trivial by the undecidability of the satisfiability of static constraints. We give in appendix a proof for nonrecursive DTDs.

*There is no (I', e') for which $(I_0, e_0), ..., (I_n, e_n)(I', e')$ is a prerun of s.

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

5.4. Workflow Constraints

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

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

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

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

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

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

Call and return guards Recall the Mail Order example, in which processing an order requires executing some tasks in a desired sequence (order, bill, pay, deliver). Since tasks in BAXML are initiated by function calls, one convenient workflow specification mechanism is to attach guards to function calls. For instance, the guard of `!Deliver`, shown in Figure 5.5, might require that the ordered product must have been paid in the correct amount. Similarly, it is useful to control when the answer of an internal function may be returned. This can be done by providing *return guards*.

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

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

A prerun $\rho = (I_0, e_0), ..., (I_n, e_n)$ of S *satisfies* $\gamma = (\gamma_c, \gamma_r)$, denoted $\rho \models \gamma$, if for each transition $I_{i_1} \vdash_{e_i} I_i$, if the transition results from a function call to $!f$ at node u the guard $\gamma_c(f)$ holds in (I_{i-1}, u), and if the transition results from the return of an internal function call $?f$ at node u, $\gamma_r(f)$ holds in $eval_{i-1}(u)$. Observe that these constraints involve consecutive instances only.

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

Example 5.8. Figure 5.5 shows call guards for some functions in the Mail Order example. The call guard of function `Bill` is given in Figure 5.2(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:

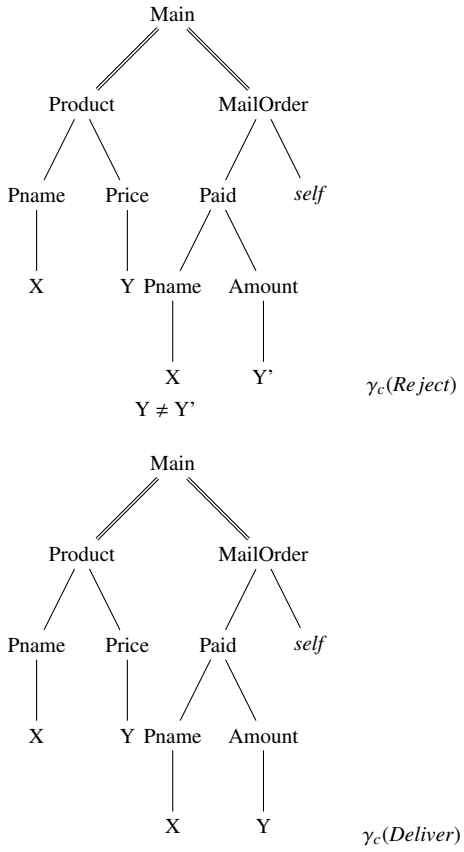
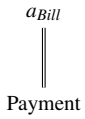


Figure 5.5.: Call guards of *Reject* and *Deliver*.



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

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

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

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

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

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

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.

An automaton for our running example is represented in Figure 5.6. The edges represent the pairs for which δ is defined, and the patterns in Υ check the following:

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

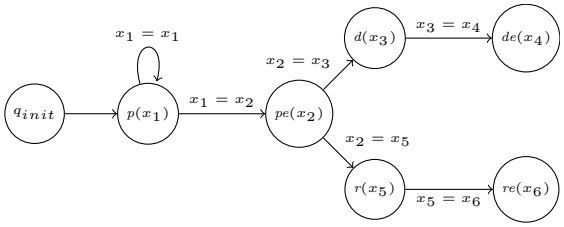


Figure 5.6.: Example of pattern automaton

We note that in some specification models, such as state-charts [Harel 87], 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 05]. We extend our formalism to capture such hierarchical notion in Chapter 6.

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 09] using only past LTL operators, that we call Past-Tree-LTL. It is obtained from classical propositional LTL (e.g., see [Emerson 90]) by interpreting each proposition as a parameterized tree pattern $P(\bar{X})$ where $\bar{X}$ is a subset of its variables, designated as *global*. All global variables are treated as free in the patterns and are quantified existentially at the end. The past temporal operators are $\mathbf{X}^{-1}$ (previously) and $\mathbf{S}$ (since), with the standard semantics. Specifically, $\mathbf{X}^{-1}\varphi$ holds for a prerun $(I_0, e_0) \ldots, (I_n, e_n)$ if φ holds at $(I_0, e_0) \ldots, (I_{n-1}, e_{n-1})$; $\varphi \mathbf{S} \psi$ holds at $(I_0, e_0) \ldots, (I_n, e_n)$ if ψ holds in $(I_0, e_0) \ldots, (I_j, e_j)$ for some $j \leqslant n$ and φ holds in $(I_0, e_0) \ldots, (I_k, e_k)$ for every k, $j < k \leqslant n$. In summary, a *Past-Tree-LTL formula* is of the form $\exists \overline{X} \psi(\overline{X})$ where ψ uses only the temporal operators $\mathbf{X}^{-1}$ and $\mathbf{S}$, and $\overline{X}$ is the set of global variables of the parameterized patterns interpreting the propositions. The set of Past-Tree-LTL formulas is denoted $\mathcal{T}$. A *TAXML schema* is an expression $S|\theta$ for S a BAXML schema and $\theta \in \mathcal{T}$. A prerun ρ satisfies $\exists \overline{X} \psi(\overline{X})$ if ρ satisfies $\psi(\nu(\overline{X}))$ for *some* valuation ν of the global variables $\overline{X}$ in the active domain of ρ.

The choice to existentially quantify the global free variables appears natural for specifying workflow transition constraints. Observe that such variables are quantified *universally* in the language Tree-LTL of [Abiteboul 09], used to specify properties of all runs. However, the model checking approach of [Abiteboul 09] is based on checking unsatisfiability of the negation of Tree-LTL formulas, whose global variables then become *existentially* quantified.

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

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

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

Checking workflow constraints

The following establishes the complexity of testing workflow constraints.

Theorem 5.9. *Let $S|W$ be a fixed workflow schema, for $W \in \{\mathcal{G}, \mathcal{A}, \mathcal{T}\}$, and ρ a prerun of S. Checking whether ρ satisfies W can be done in* PTIME *with respect to $|\rho|$.*

A more difficult decision problem is checking the *existence* of a valid transition extending the current prerun. Indeed, this is undecidable even for BAXML schemas with no workflow constraints (with either flavor of the abort semantics). The difficulty arises from the power of external functions. This cannot be done for external functions because the set of possible answers is typically infinite, and of arbitrary depth. Indeed, without external functions it suffices to test all possible call activations and returns. However, the problem becomes decidable for bounded trees.

Theorem 5.10. *(i) Given a BAXML schema S and a prerun ρ of S, it is undecidable whether ρ is blocking. (ii) Given a BAXML schema S with non-recursive DTD and a prerun ρ of S, it is decidable whether ρ is blocking.*

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

5.5. Expressiveness

In this section we compare the expressive power of BAXML, GAXML, AAXML, and TAXML, using the framework developed in Section 5.2. 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 5.2. For each specification S (for BAXML) or $S|W$ (for GAXML, AAXML and TAXML), the nodes of the workflow system are the finite prefixes of runs of S or $S|W$. The state label for each node is the last instance in the prefix. The root is the empty prerun, denoted Φ. There is an edge labeled e from node ν to ν' if ν' extends ν with a single instance by event e that is a function call or the return of a such a call. Note that the infinite paths in the tree starting from Φ correspond to the runs of $S(|W)$. Because of the semantics of blocking runs, each path is extensible to an infinite path.

Note that there are alternative 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.

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 5.4, the existence of simulation with respect to any coarser view. Unfortunately (but not surprisingly), the three extensions of BAXML models are incomparable relative to the identity view.

Theorem 5.11. *The workflow languages GAXML, AAXML and TAXML are incomparable relative to* $\hookrightarrow_{(id,id)}$.

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

Figure 5.7.: Cost of various simulations from Theorems 5.12 and 5.13

Comparison with projection views

Given the negative result of Theorem 5.11, we next consider simulation relative to views allowing more leeway in the simulating system. Specifically, the view remains the identity on the simulated system, but allows the simulating system to use additional data and functions. We refer to the latter as a projection view and denote the class of projection views by π.

Specifically, let S be a BAXML schema and Σ_0 and $\mathcal{F}_0$ be subsets of the tags and functions of S (the visible symbols) such that, in every instance satisfying the DTD of S, whenever a node has tag $a \notin \Sigma_0$, none of its descendants has a label in Σ_0 or in $\mathcal{F}_0$. The projection $\pi_{\Sigma_0,\mathcal{F}_0}([S])$ 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 $!f(F)$ is ε for $f \notin \mathcal{F}_0$ and $!f(\pi_{\Sigma_0,\mathcal{F}_0}(F))$ for $f \in \mathcal{F}_0$, and similarly for $?f(F)$. The projection view is defined in the same way for BAXML augmented with constraints (GAXML, AAXML, and TAXML).

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

Theorem 5.12. $\mathcal{W} \hookrightarrow_{(id,\pi)}$ BAXML
for $\mathcal{W} \in \{\text{GAXML}, \text{TAXML}^{sib}, \text{AAXML}^{sib}\}$.

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

For AAXML and TAXML, we have the following.

Theorem 5.13. *AAXML* $\hookrightarrow_{(id,\pi)}$ *TAXML and*
TAXML $\hookrightarrow_{(id,\pi)}$ *AAXML.*

The proofs of the above results (see appendix) provide insight into the simulations of the four languages by each other, and in particular into the power of imposing control using static constraints. In terms of the cost of each simulation, several parameters can be considered:

- the blowup in the schema size,
- the blowup in the instance size,
- 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 5.7.

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

Theorem 5.14. $\mathcal{W} \not\hookrightarrow_{(id,\pi)}$ *GAXML*
for $\mathcal{W} \in \{\text{TAXML}, \text{AAXML}\}$.

Comparison with coarser views

Theorem 5.12 shows that BAXML, GAXML, TAXML^{sib} and AAXML^{sib} can simulate each other relative to projection views. This result turns out to be quite powerful. Indeed, by Lemma 5.4 it implies that the simulation results can be extended to any views that are coarser than such views. For example, one may wish to focus on the sequence of events (function calls and returns, together with their arguments), ignoring state information. This information can be captured by composing the views in id and π with a view V that is the identity on events and maps every state to a fixed constant. By Lemma 5.4, BAXML, GAXML, TAXML^{sib} and AAXML^{sib} can simulate each other relative to $(id \circ V, \pi \circ V)$. Similar remarks apply to TAXML and AAXML.

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

5.6. BAXML and Tuple Artifacts

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

We first review informally the tuple artifact model, as presented in [Deutsch 09] (see Subsection 5.6.1 for the formal definition). We denote the model by $\mathcal{TA}$. We assume an infinite data domain D. An artifact system consists of a set of artifacts and a set of services acting on the artifacts. An artifact consists of an *artifact tuple* and a set of *state relations*. In addition, an artifact system has an underlying database shared by all artifacts and services, that is fixed throughout a run of the system.

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

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

Services are applied non-deterministically. At any given time, a service can be applied to the current instance if its pre-condition holds and if the post-condition is satisfiable. Thus, there are two forms of non-determinism in a transition: one stemming from the choice of service, and another from the choice of values for the next artifact tuples, from among those satisfying the post-condition.

A *run* of an artifact system is a sequence of consecutive instances together with the name of the service applied at each transition. (For initial instance, we take any instance whose artifact states are empty.) As for BAXML, blocking runs are extended by repeating forever the last configuration, with the corresponding transitions labeled by the special event *block*. See [Deutsch 09] for a detailed example of an artifact system.

5.6.1. The Tuple Artifact Model

We provide the definition of the tuple artifact model, adapted from [Deutsch 09]. A relational database schema $\mathcal{D}$ consists of a finite set of relation symbols with specified arities. The arity of relation R is denoted $a(R)$. 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 $a(R)$. 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, by default 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 5.15. 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 $C = \langle \mathrm{R}, \mathrm{S} \rangle$, 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 5.16. An *artifact schema* is a tuple

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

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

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

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

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

Definition 5.17. An *instance* of an artifact schema

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

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

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

Definition 5.18. A *service* σ over an artifact schema $\mathcal{A}$ is a tuple $\sigma = \langle \pi, \psi, \mathbf{S} \rangle$ where:

- π, called *pre-condition*, is a sentence in $\mathcal{L}_\mathcal{A}$;
- ψ, called *post-condition*, is a formula in $\mathcal{L}_\mathcal{A}$, with free variables $\{\bar{x}_R \mid R \text{ is an attribute relation of a class in } \mathcal{A}\}$;
- $\mathbf{S}$ is a set of *state rules* containing, for each state relation S of $\mathcal{A}$, one, both or none of the following rules:
  - $S(\bar{x}) \leftarrow \varphi_S^+(\bar{x})$;
  - $\neg S(\bar{x}) \leftarrow \varphi_S^-(\bar{x})$;

  where $\varphi_S^+(\bar{x})$ and $\varphi_S^-(\bar{x})$ are $\mathcal{L}_\mathcal{A}$-formulas with free variables $\bar{x}$ s.t. $|\bar{x}| = a(S)$.

Definition 5.19. An *artifact system* is a pair $\Gamma = \langle \mathcal{A}, \Sigma \rangle$, where $\mathcal{A}$ is an artifact schema and Σ is a non-empty set of services over $\mathcal{A}$.

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

1. $A \models \pi$;
2. $A'|\mathcal{DB} = A|\mathcal{DB}$;
3. if $\bar{u}_R$ is the content of the attribute relation R of $\mathcal{A}$ in A', then A satisfies the post-condition ψ where $\bar{x}_R$ is replaced by $\bar{u}_R$ for each R;
4. for each state relation S of $\mathcal{A}$ and tuple $\bar{u}$ over $adom(A)$ of arity $a(S)$, $A' \models S(\bar{u})$ iff

$$\begin{aligned} A \models \quad & (\varphi_S^+(\bar{u}) \wedge \neg\varphi_S^-(\bar{u})) \vee (S(\bar{u}) \wedge \varphi_S^+(\bar{u}) \wedge \varphi_S^-(\bar{u})) \\ & \vee (S(\bar{u}) \wedge \neg\varphi_S^+(\bar{u}) \wedge \neg\varphi_S^-(\bar{u})) \end{aligned}$$

   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.

Note that, according to (2) in Definition 5.20, 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 so 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 $\Gamma = \langle \mathcal{A}, \Sigma \rangle$. An *initial instance* of Γ is an artifact instance over $\mathcal{A}$ whose states are empty.

Definition 5.21. A *prerun* of an artifact system $\Gamma = \langle \mathcal{A}, \Sigma \rangle$ is a finite sequence $\rho = \{(\rho_i, \sigma_i)\}_{0 \leqslant i \leqslant n}$ where each ρ_i is an artifact instance over $\mathcal{A}$ and each σ_i is a service, such that:

- ρ_0 is an initial instance of Γ;
- $\sigma_0 = \mathit{init}$;
- 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 $\{(\rho_i, \sigma_i)\}_{i \geqslant 0}$ 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*.

5.6.2. Comparison

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

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

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

Theorem 5.22. $\mathcal{TA} \hookrightarrow_{(\mathrm{id}, \mathcal{V}_{\mathcal{TA}})}$ *BAXML.*

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

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

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

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

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

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

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

Theorem 5.23. *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.*

The proof (using only BAXML schemas of bounded depth) relies on the undecidability of implication for functional and inclusion dependencies (see appendix).

Remark 7. By Lemma 5.4 (applied to effective simulations), the negative result of Theorem 5.23 extends to any views that expose *more* information than those above.

Chapter 6.

An implementation of the AXML artifact model: AXART

6.1. Introduction

This chapter extends the core of the AXML Artifact model, e.g. a BAXML schema possibly with some workflow constraints, presented in the previous chapter. This chapter proposes the AXART system as a platform for collaborative work in a centralized environment. The AXART system is based on the AXML Artifact model.

We extend the AXML Artifact model with several interesting features: control of the interactions with the user, dynamic modification of the workflow ,and distribution. The dynamic modification of the workflow and the management of access rights are new with respect to previous works. We illustrate these with a real-life example: applying for a movie role in the movie industry, as described in [Wikipedia]. This is representative of widely occuring cases where a workflow is owned by a company or a public institution and with its partners. The example shows how a standard workflow is modeled with AXML Artifact and how users can easily interact with the artifacts using forms and modifying the states of the artifacts, effectively running the workflow. The artifact changes are monitored by the AXML system and notifications are sent to potentially concerned users, so that they can take appropriate actions. The example also illustrates the dynamic modification of the workflow by two mechanisms, namely *workflow specialization* and *workflow exception*, that allow users with proper rights to modify the workflows during their runs. This provides useful flexibility in the workflows built with the AXML Artifact model. The changes on the data and the workflow of an artifact are governed by the security rights that a user has for that artifact. The distribution is illustrated by an application to Dell Supply chain [Kapuscinski 04].

The AXART system is a system managing centralized human interactions. It implements a restriction of the AXML Artifact model. For example, the only allowed function are external functions where the returned trees represent the information filled by a user. The main idea of the implemented rules is to represent the workflow as data rules involving queries on the documents. The system AXART maintains dynamically the views needed to verify the evolution of the workflow. It uses monitoring techniques such as view maintenance on (active) documents in the presence of positive updates. AXART combines these techniques with security techniques for managing access rights. This allows the management of a large collection of artifacts with data access control.

The chapter is organized as follows. In the next section, we present two motivating examples. In Section 6.3, we present briefly the extensions to the core of AXML Artifact model presented in the previous Chapter. Section 6.4 presents the AXART system.

6.2. Two motivating examples

We consider here two examples, illustrating the different extensions that we present in this chapter. The first example illustrates the following extensions: hierarchy of artifacts, dynamic modification of the workflow and access control. The second illustrates the distribution extension.

Role Application Procedure

The Setting We consider as motivating example the *Role Application Procedure* in Hollywood. It is a workflow owned by a Film Company, whose purpose is to cast actors for a movie. There are typically four types of participants involved in such a workflow: the *Actor*, the *Casting Panel*, the *Casting Assistant* and the *Film Director*. Each of them corresponds to a security role in our system.

A Film Director creates an application to deal with the casting of his movie. For each role, he creates a subtask. The actors can apply to a role by creating a sub artifact the *Film Role Application* (FRA).

The Casting Assistant filters the applications e.g., eliminates the ones that are not filled in properly and schedules auditions for the Film Actors. At the due date, the audition takes place in front of two Casting Panels. It might result in the rejection of the application or in a "passed audition" verdict. This procedure is a standard version of the workflow, as defined initially for the class of FRAs. In addition, the Casting Panels may suggest new auditions and in this case details need to be provided by the Casting Panels for organizing them. This is a case of Workflow specialization.

A "passed audition" verdict means that the Actor is eligible for the role and that its application will be considered by the Director. Out of a pool of FRAs that passed auditions, the Director will choose one. Of course, the Director might have a short list of favorite actors and could make his choice at any moment if he spots the an application of a favorite actor, even without having the actor passing an audition for the role. This is a case of Workflow exception.

Each task is represented by an artifact. In our context of centralized system, the users log in the system and all their actions are directly supervised by the system.

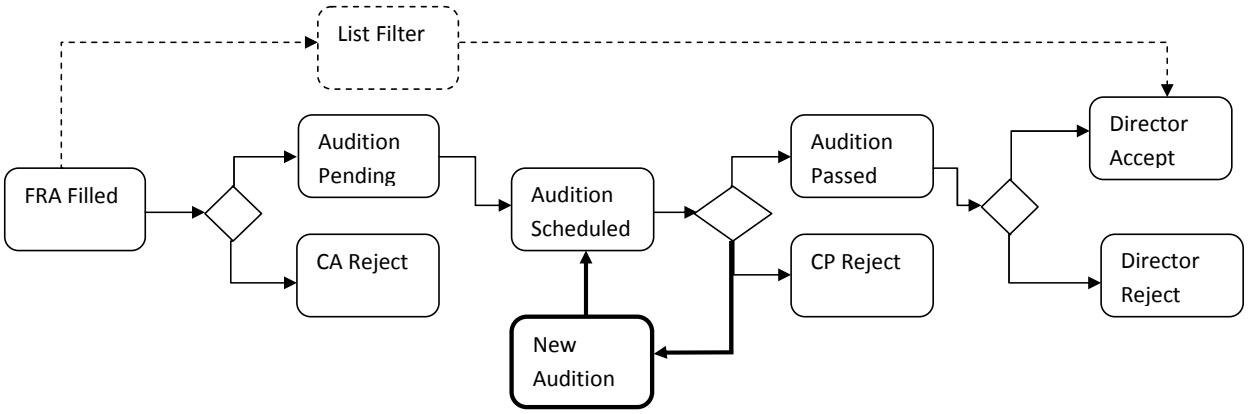


Figure 6.1.: The standard workflow of a FRA (solid), its exception (dotted) and its specialization (bold)

An example Scenario We start with the standard workflow (the regular boxes and arrows in Figure 6.1) that is pre-defined for an artifact of class FRA. The boxes represent stages in the workflow, while the diamonds stand for choice operators. In the case of the standard casting workflow we suppose that only an audition needs to be passed. At any moment, the artifact will be viewed by the workflow participants as a form with fields to fill in. This figure represents the unfolding of the workflow of the FRA and the workflow of the audition associated to it.

In the first part of the scenario, the Actor *John Travolta* fills the attached FRA with details about him. When it is done, an audition is scheduled by a Casting Assistant. When the Casting Panel logs in, it will be able to choose one of the FRAs in the stage Audition Scheduled. When choosing the FRA of John Travolta, it will need to fill in a form as the one in Figure 6.2. Supposing the two decisions of the Casting Panels are positive, the FRA of John Travis will be considered by the Director at his next log in.

In a second time, we illustrate the principle of Workflow exception. The *Director* may have a list of preferred actors for the main role. The Director logs in first. He will be able to edit a list of his preferences for the Role *Batman* in the film *A Batman Romance*. He will place the actor *John Travolta* as his first preference in the list and he will log out. This is a Workflow Exception. The dotted arrow and box in Figure 6.1 shows the exception when the Director chooses to approve John Travolta's application at the first step. When the application of John Travolta matches the list of preferences, i.e. it enters the *List Filter* stage, the Director is notified that should log in and explicitly validate it.

The third part instances how the workflow can be changed by Workflow specialization during the run of the application. The Casting Panel wants the actor *John Who* to pass another audition. The bold arrows and boxes show the extension of the workflow.

Dell Manufacturing system

To illustrate the distribution aspect, we use as a running example a simplified view of the Dell manufacturing system [Kapuscinski 04]. See Figure 6.3.

The manufacturing system processes continuous flows of orders and has to cope with issues such as distant suppliers. The main modules are as follows. The WWW interface is the Webstore, in charge of processing forms completed by customers and of generating orders to the dispatcher. For a given order, Dispatch selects a plant close to the customer to delegate order processing. Each Plant processes an order, by forwarding orders for different parts to the relevant warehouses. It processes an order, by combining the parts that are received into objects (e.g. computers) that are then physically sent to the customers. Warehouse is a platform acting as a buffer between suppliers and Dell's plants. Finally, Credit Service is a third party in charge of checking the validity of credit card payments when an order is created at a Webstore.

In the running example, when a new Web order arrives (1), a new *webOrder* artifact is created. Then the new artifact creates a subartifact that is sent to a credit service (2). Once credit has been approved, the subartifact returns to the *webOrder* but now its state contains all the credit data (and notably the fact that the credit request was successful). A plant is then selected and the artifact moves to that plant (3). It initiates a new subartifact for gathering parts, that is sent to a warehouse and another local artifact for communications with the customer (4). Once the product has been built, the artifact is sent to a delivery service (5). Finally, once the Web order has been completed, the artifact moves to an archive where it is stored as a text-based XML serialization that includes all the information it has gathered during its life cycle (6).

Figure 6.2.: Form snapshot for a stage (Audition Scheduled)

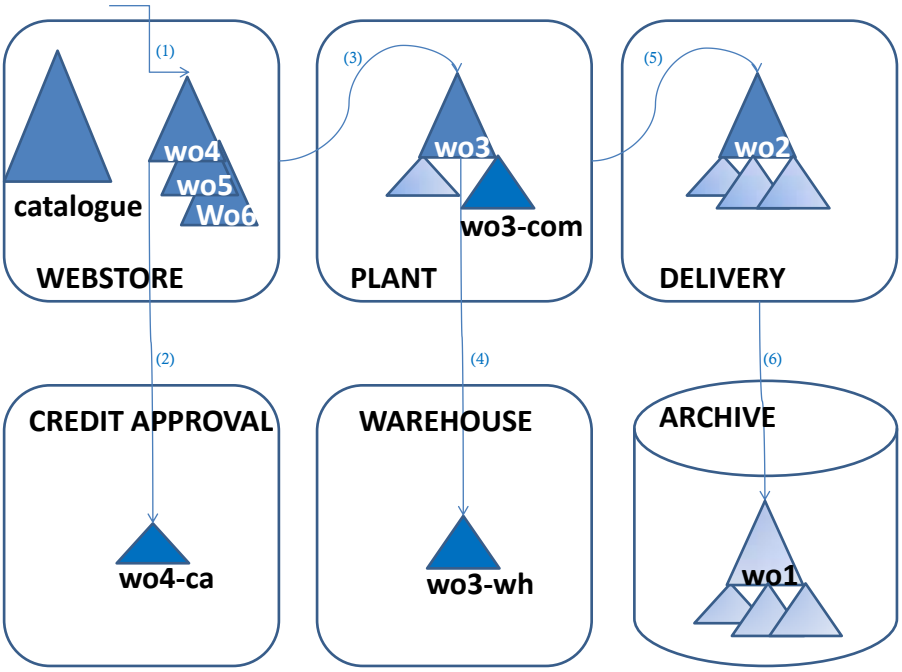


Figure 6.3.: Artifacts in the Dell application

6.3. Extensions of the AXML Model

Hierarchy of artifacts

In real workflows, artifacts can be related to each other in a hierarchical. In the AXML Artifact model, an artifact can have *subartifacts* that are considered as its subtasks. In the movie example, each task managing a role may have several subtasks managing the applications of the actors. We materialize this relation between artifact and subartifact using active documents with subtrees. A artifact α is a subartifact of the artifact β iff the active document representing α is a subtree of the active document representing β. Each (sub)active document representing an (sub)artifact is identified by the tag *Artifact* labeling its root. Moreover, we assume that each node labeled *Artifact* has a child labeled *Id* which has an associated value. This value is the identifier of the artifact, which is assumed unique. An artifact α is an *ancestor* of the artifact β iff the active document of α has the subtree of β.

Moreover, we equip each (sub)artifact with its own schema. An action of an artifact is not only constrained by the workflow of its schema but also by the workflows of its ancestor(s). Indeed, when an action is made for an artifact, it changes also the active documents of its ancestors that have to satisfy their workflow constrains. In the rest of the chapter, we consider that the workflow constraints are expressed by Tree pattern automata, but other constraints may be used. Remark that a function can return an artifact or classical active documents.

We now extend the schema to allow access control, workflow modifications and distribution.

Access Control

In the scenario, the notion of user activating a call is introduced. Users can activate function calls only if they have rights to do so based on their roles. We associate to each state of the Tree pattern automata a set of pair (*role name*, *function name*) with the following semantics: the user with the role r can activate the function call $!f$ iff the pair (r, f) is associated to the current state of the artifact. Remark that now, an activation of a function call is constrained by the workflow constraints and by access controls.

Dynamic modification of the behavior of an artifact

Our system allows to modify the authorized behaviors of an artifact principally based on the modification of the schema of the artifact. We distinguish two kinds of modifications : *specializations* that constraint the possible future behaviors of the artifact, and *exceptions* that relax the possible future behaviors of the artifact. These workflow extension mechanisms are new features proposed in this chapter. Specialization is related to Business rules for the tuple artifact model introduced in [Bhattacharya 07c] and studied also [Deutsch 09]. Business rules are conditions that can be super-imposed on the pre-conditions of existing in order to customize their behavior services without changing their implementations.

A *specialization* is based on a modification of the schema and by adding possible new subartifacts. The new schema has to be a restriction of the previous schema: the static constraints have to imply those of the previous schema. In the same way, the workflow constraints have to be more constrained. We impose the following syntactic constraints. The two workflow constraints have to be defined over the same set of states. For each constraint, the formulas of the new workflow has to imply the old one (by taking into account the static constraints) and the new function δ has to be more refined than the old one. A specialization may add some subartifacts that were not allowed to add even by activating function calls. This is for instance the case in the movie example when a new audition has to be scheduled after the first one. In the main workflow, this possibility was not planned and no function calls may bring a new audition artifact. The only way to schedule one would be by specializing the artifact.

Similarly, *exceptions* are based on the modification of the schema. In the case of an exception, the new schema relaxes the constraints. The new static constraints are more general. The access controls may also be a relaxed as well as the workflow constraints. The states are the same but the transition function is more general as well as the formulas of the states. For instance, in the movie example, the Director accepts an application if the actor passed the auditions successfully. In the second scenario, the Director relaxes the constraints for John Travolta. The Director accepts an application if the actor passed the audition successfully, or is called John Travolta.

These kinds of modifications can be done only if after the change the artifact and its ancestors may remain in their current stage and the user is authorized to make it. As for the activation of functions, rights for each kind of action are associated to a role and a state .

Distribution

In a distributed environment, it might be the case that artifacts are processed outside of the system that originated them. This kind of semantics is not considered in the previous chapter. In particular, a subartifact may be moved outside. Such an artifact contains the set of rules that define its valid evolution with respect to the workflow, so these rules can be enforced by the partner systems. However, there is no guarantee that the partner system will not violate rules. In the example of

the Film Role Procedure, the Impresario fills out the application for the role outside the Film Company's system, be awarding of the rules that need to be followed. In our model, an artifact may leave and then return into the secure system. The system only gives the artifact to a user who is authenticated and has the proper access rights. When the artifact returns to the secure system, the system checks that the user modified the artifact according to the rights associated with his role. The received artifact will be rejected otherwise. Observe that in a distributed variant of our scenario all the participants, except for the Impresario, log in and work in the secure environment provided by the system of the Film Company. Checking the evolution of an artifact against security rights is not the topic of this thesis and is not supported by the AXART system.

6.4. The AXART System

6.4.1. The implemented Model

The model implemented in the AXART system is a restriction of the AXML artifact model that focuses on the human interactions. A user can interact with the system by activating a function call. Then he has to fill a form representing the tree returned to system. In this way, the user has both client and server interactions with the AXART system.

In the system, an active document is represented by an AXML document. See Figure 6.4 where the tree is represented using an XML syntax (a text-based serialization of the tree). The figure shows part of a Film Role artifact immediately after the activation and the return of the function call *Artifact0.Init*. The function call *Artifact0.newFilmRoleApplication* is activated next in order to create a FilmRoleApplication artifact. All functions are external. Each function DTD can only represent a fix tree with possible data values representing the fields that a user has to fill.

```
< Artifact >
< Content >
    < axml:sc xmlns:axml="http://futurs.inria.fr/gemo/axml/" axml:id="Artifact0.Init" id="n2" >
    < FilmRoleApplications >
      < axml:sc xmlns:axml="http://futurs.inria.fr/gemo/axml/" axml:id="Artifact0.newFilmRoleApplication" >
    < /FilmRoleApplications >
    < RoleName > PulpFiction < /RoleName >
  < /Content >
    ...
  <  <ArtId id > Artifact0 < /ArtId >
< /Artifact >
```

Figure 6.4.: An AXML artifact

In order to simplify the implementation, we require that there are no two calls to the same service in a same active document. This is done by imposing syntactic constraints on the DTDs. The right to activate a function depends on the role of the user and on the state of the artifact.

To ensure good performance in our system, we impose two restrictions:

1. Tree pattern query nodes are not labeled by function tags.

2. The activation and the return of a function form an atomic action for our system, in particular there are no internal function calls.

Moreover, we impose syntactic restrictions on the workflow exceptions and the workflow specifications. We assume that the Boolean combination of tree-patterns are in disjunctive form. A specialization of a formula φ can only be made by adding conjuncts to a conjunction of φ. Adding artifacts for a specialization can only be done under a specific label *SchemaUpdate* labeling only one child of the artifact's root. An extension of a formula φ can only be made by adding disjunction to it.

To summarize, in our system, an artifact is at any given time an active document (or part of it). It contains its data and the data of its subartifacts, as well as information about the workflow stages the artifact and its subartifacts are in. An artifact can be in a stage if it satisfies the formula associated to its current state. Finally, the workflow of a schema describes the possible transitions. Because we do not impose that an artifact should satisfy only one stage formula at a time, it is possible to change the stage without updating the content of the artifact. An artifact evolves when a user calls a function of the AXML document. Following the activation of a function call, the same user fills the fields of the function. The update is accepted if it leads to valid workflow stages for all artifacts.

6.4.2. Algorithm

The main technical difficulty of the system is to compute in runtime the possible functions that may be activated by a user following his role and the current stage of this artifact. to this end, we look for the possible trees that a user may return without violating the constraints of the workflows associated to the current artifact and its ancestor. Remark that the structure of a tree returned by a function is fixed, only its values can change. So, we look for the possible valuations that a user can return. Because of the restrictions, we can assume that the model is monotone and we can use techniques and notions presented in Part I. For each function call f and tree-pattern query appearing in a formula, we compute the *scenarios*, see Definition in Section 2.6. Remain that a scenarios explain how a set function call may contribute to the satisfaction of a tree-pattern query. We look for the scenario where only f appears the provenance tuple. From these scenarios and the definition of the return tree of f, we extract generalized tuples representing the allowed valuations of the variables of the returned tree of f. The generalized tuples are manipulated through the operators used to define the formula. A function can be activated iff the user has the right to activate the function and if the set of the possible valuations is not empty. These structures are maintained incrementally when the artifact is updated.

6.4.3. The Architecture

The main goal of the system is to manage the updates of artifacts. An update waits to be applied until the Artifact Manager checks that the artifact and the higher-level artifacts can transition to a next stage under this update. The properties of stages are expressed by Boolean combinations of Patterns that have to be checked. The Artifact Manager is helped by the Axlog Module that maintains incrementally the answers and the scenarios of the tree-pattern queries used to define the properties of the states, see Section 2.6. The AXART system is built on top of the AXML system, used as AXML DocStore in Figure 6.5, that is a manager of active documents. The update of an AXML document is done through a service of the AXML system.

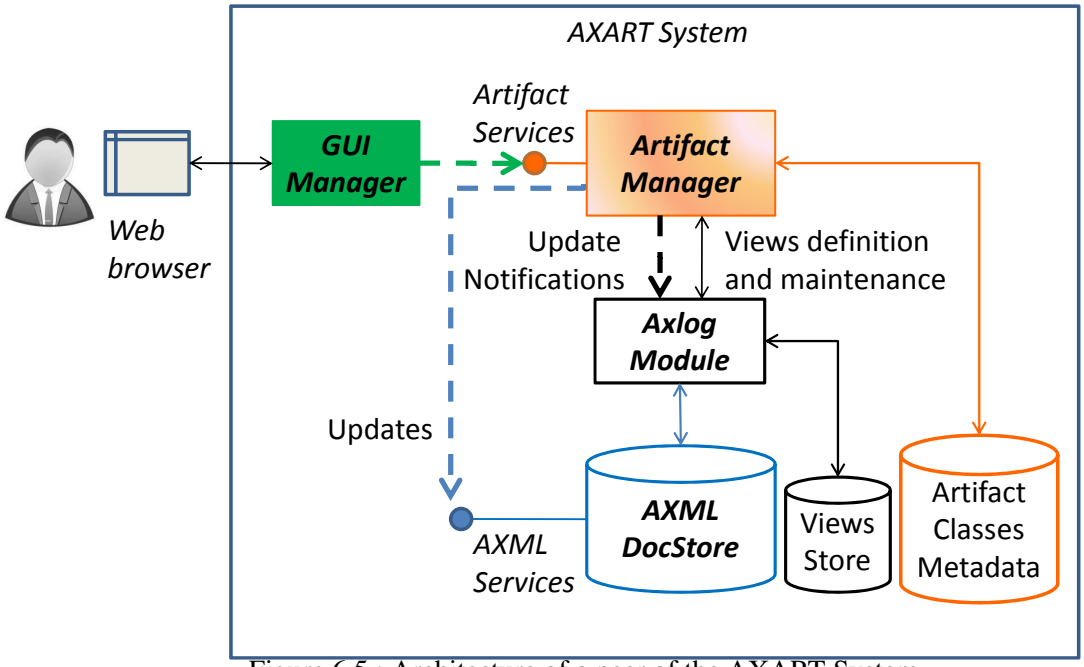


Figure 6.5.: Architecture of a peer of the AXART System

Access control and monitoring queries are handled at the peer level. The architecture of an AXART peer is presented in Figure 6.5.

The system maintains values of the stage formulas as views, using Axlog widget, see Chapter 3 and for more details [Marinoiu 09]. Thanks to a subscription service that we developed in Axlog, the AXART system would be easily extended to allow users to receive alerts about artifacts that concern them. For instance, the Director can register into the system and ask notified whenever a favorite actor applies for the role of Batman. Similarly, an Actor can register a task to the monitoring system and be notified whenever one of his applications has been approved. The monitoring of artifacts model are discussed in [Marinoiu 09]

6.4.4. The User Interface

The system provides to the users with a Web interface for managing the artifacts. After logging into the system, the user has access to the set of artifacts that he can modify, has sent or received. He also has a list of functions that he can activate according to his security rights.

When a user chooses to activate a function, a form as in Figure 6.2 is presented to him. The system may propose the set of possible valuations, which are dynamically generated, to the user. In Figure 6.2, the user can choose one of the *Accepted*/*Rejected* options. He can create or import artifacts from other systems, e.g. received by email.

In order to modify the workflow, the system provides a high-level language that supports the creation of an artifact of a known class. The user may redefine the constraints of stages using a set of basic properties suggested by the system. Defining new properties requires an expert, since tree-pattern queries are involved.

Conclusion

We study in this thesis how active documents can provide the basis for the management of distributed data. We focus on two kinds of data management systems: stream processing and data-centric workflows.

Stream Processing: the Axlog model

Our first contribution is the study of active documents for stream processing. Works on stream processing have focused on processing intensive streams where the relevant data arrives in a short time. We study here stream processing in a different setting where relevant data arrives over long periods of time. In this context, we propose a new stream processor model called Axlog widget, specified using a query over an active document. We developed efficient techniques to maintain queries over active documents. These techniques are based on two theoretical notions: satisfiability and relevance. In particular, they allow combining techniques used for database theory and stream processing. We show the tractability of satisfiability and relevance notions for monotone Axlog widgets. The algorithm to maintain queries over active documents and the use of Axlog widgets are fully developed in [Marinoiu 09]. System issues are briefly presented in this thesis to motivate the study of satisfiability and relevance.

Data centric workflow: the AXML Artifact model

Our second contribution is the study of active documents used in the context of data-centric workflows. Several previous models based on active active documents have been proposed in [Abiteboul 09] and [5]. These works focused on different aspects of the workflows, the model checking for [Abiteboul 09] and a global distributed model for [5]. In this thesis, we revisit the AXML model, in order to understand and compare the different workflow constraints introduced in [Abiteboul 09], based on guards for activation and return of function calls and in [5], based on automata. To this end, we introduce a common core of these models called BAXML. We then augment BAXML with different workflow constraint languages. The contribution of the thesis is dual. 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 are first investigated using BAXML. We prove 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 relationship to static constraints. Finally, we compare BAXML to tuple artifacts, a variant of IBM's Business Artifact model using relational databases. We show that BAXML can simulate tuple artifacts whereas the converse is false. To compare these very

different models, we use again the views framework to render them compatible. This illustrates the usefulness of the view-based framework to reconcile seemingly incomparable workflow models.

Then, we extended BAXML coupled with workflow constraints with new features or some presented in [5] to obtain the AXML Artifact model. The AXML Artifact model supports description of workflow constraints, hierarchy of tasks, access control, worklow update and distribution. We have implemented a prototype, AXART based on the extended AXML Artifact model. The prototype implements a subset of the AXML Artifact model, focusing on human interactions.

Perspectives

Data centric workflows provide a challenging topic where much work remains to be done. We mention next three future research directions.

The first is the extension of AXART to a distributed setting in order to fully support the model proposed in [5]. The new system will have to deal with checking access control in a distributed environment. A first idea would be to see how techniques developed in [Abiteboul r] and [Abiteboul f] and demonstrated in [Antoine 11] could be adapted to this context. Another approach which seems particularly promising would be to check access controls a posteriori. For example, an artifact is sent from Alice to Bob, who is an untrusted peer. After receiving back of this artifact from Bob, Alice has to check if Bob correctly obeyed the access rights. Watermarking techniques can also be used this problem.

The following two points are based on the view framework for comparing workflows. First remark that this framework allows to provide several explanations of the same workflow based on different workflow languages. The workflow semantics is a tree of runs described in a particular formalism. Its execution can be explained in a different formalisms for a user using the notion of view based simulation. For example, an AXML artifact having a BAXML schema may have an explanation using a AAXML schema (the BAXML schema is a simulation of the AAXML schema) that would be more readable for the user.

The second topic is to describe powerful interfaces of systems, in particular choreography contracts between two systems. It seems that that view-based simulations and artifacts would be particularly suitable for this task. An interface would be an artifact describing possible and obligatory behaviors of its implementations. An artifact would implement another artifact by using a relation based on views for simulation. The notion of view simulation would be adapted to define the implementation relation. It would be a way to describe powerful contracts with data at the center, in particular for applications where privacy is essential, like in social network applications.

The last topic is in line with model checking of AXML artifacts. In previous work, each restriction of the model needed to obtain decidability consists in trading off part of recursion in the process, non-monotonicity and unbounded data. One could also consider combining different techniques with abstractions of complex artifacts. Some behaviors of the artifact would be abstracted to check particular properties. With a “right“ notion of abstraction, based on views, the properties checked on the abstraction would be also true for the “real” artifact.

Part III.

References

Self References

Journal Article

[1] Claude Jard, Thomas Chatain and Pierre Bourhis Diagnostic temporel dans les systèmes répartis à l'aide des dépliages des réseaux de Petri. *JESA*, 39:351– 366, 2005.

Conference Articles

[2] Serge Abiteboul, Pierre Bourhis and Victor Vianu. Comparing Workflow Specification Languages: A Matter of Viewso In *Proc. ICDT*, Uppsala, Sweeden March 2011.

[3] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Satisfiability and Relevance for Active Documents. In *Proc. PODS*, Providence, USA, June 2009.

[4] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Efficient maintenance techniques for views over active documents In *Proc. EDBT*, Saint Petersburg, Russia, March 2009.

[5] Serge Abiteboul, Pierre Bourhis, Alban Galland and Bogdan Marinoiu The AXML Artifact Model. In *Proc. TIME*, Brixen-Bressanone, Italia, July 2009.

[6] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Incremental View Maintenance of Active Documents. In *Proc. BDA*, Marseilles, France, October 2007.

[7] Claude Jard, Thomas Chatain and Pierre Bourhis. Diagnostic temporel dans les systèmes répartis à l'aide des dépliages des réseaux de Petri. In *Proc. Modélisation des systèmes répartis*, Autran, France, October 2005.

Demonstrations

[8] Serge Abiteboul, Pierre Bourhis, Alban Galland and Bogdan Marinoiu Axart, Enabling Collaborative works with Active XML Artifacts. In *Proc. VLDB*, Singapur, Singapur, September 2010.

[9] Serge Abiteboul, Bogdan Marinoiu and Pierre Bourhis. Distributed Monitoring of Peer-to-Peer Systems. In *Proc. ICDE*, Cancun, Mexico, April 2008

Bibliography

[Abadi 03] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul & S. B. Zdonik. *Aurora: a new model and architecture for data stream management.* VLDB J., vol. 12, no. 2, 2003.

[Abadi 05] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing & S. B. Zdonik. *The Design of the Borealis Stream Processing Engine.* In CIDR, 2005.

[Abiteboul 95] Serge Abiteboul, Richard Hull & Victor Vianu. Foundations of databases. Addison-Wesley, 1995.

[Abiteboul 96] S. Abiteboul, L. Herr & Jan Van den Bussche. *Temporal Versus First-Order Logic to Query Temporal Databases*. In Proc. ACM PODS, pages 49–57, 1996.

[Abiteboul 98] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos & Janet L. Wiener. *Incremental Maintenance for Materialized Views over Semistructured Data.* In VLDB, pages 38–49, 1998.

[Abiteboul 00] S. Abiteboul, V. Vianu, B.S. Fordham & Y. Yesha. *Relational Transducers for Electronic Commerce*. JCSS, vol. 61, no. 2, pages 236–269, 2000. Extended abstract in PODS 98.

[Abiteboul 04a] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo & Nicoleta Preda. *Lazy Query Evaluation for Active XML.* In SIGMOD Conference, pages 227–238, 2004.

[Abiteboul 04b] Serge Abiteboul, Omar Benjelloun & Tova Milo. *Positive Active XML.* In PODS, pages 35–45, 2004.

[Abiteboul 06] Serge Abiteboul, Luc Segoufin & Victor Vianu. *Representing and querying XML with incomplete information*. ACM Trans. Database Syst., vol. 31, no. 1, pages 208–254, 2006.

[Abiteboul 07] Serge Abiteboul & Bogdan Marinoiu. *Distributed Monitoring of Peer to Peer Systems*. In Workshop On Web Information And Data Management, pages 41–48, 2007.

[Abiteboul 08a] Serge Abiteboul, Omar Benjelloun & Tova Milo. *The Active XML project: an overview*. VLDB J., vol. accepted for publication, 2008.

[Abiteboul 08b] Serge Abiteboul, Ioana Manolescu & Spyros Zoupanos. *OptimAX: efficient support for data-intensive mash-ups*. In ICDE, pages 1564–1567, 2008.

[Abiteboul 09] Serge Abiteboul, Luc Segoufin & Victor Vianu. *Static analysis of Active XML systems*. In TODS, 2009. Extended abstract in PODS 08.

[Abiteboul 11] Serge Abiteboul, Balder ten Cate & Yannis Katsis. *On the Equivalence of Distributed Systems with Queries and Communication*. In ICDT, 2011.

[Abiteboul r] Serge Abiteboul, Meghyn Bienvenu, Alban Galland & Marie-Christine Rousset. *Distributed Datalog Revisited*. Datalog 2.0 Workshop, 2010 (To appear).

[Abiteboul f] Serge Abiteboul, Alban Galland, Amélie Marian & Alkis Polyzotis. *WebdamExchange, a Model for Data Access on the Web*. In preparation, draft on `http://webdam.inria.fr/drafts/WebdamExchange.pdf`.

[Adam 98] N. Adam, V. Atluri & W. Huang. *Modeling and analysis of workflows using Petri nets*. Journal of Intelligent Information Systems, vol. 10, no. 2, pages 131–158, 1998.

[Alon 03] Noga Alon, Tova Milo, Frank Neven, Dan Suciu & Victor Vianu. *XML with data values: typechecking revisited*. JCSS, vol. 66, no. 4, pages 688–727, 2003.

[Alur 05] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps & Mihalis Yannakakis. *Analysis of recursive state machines*. ACM Trans. Program. Lang. Syst., vol. 27, no. 4, pages 786–818, 2005.

[Antoine 11] Emilien Antoine, Alban Galland, Kristian Lyngbaek, Amélie Marian & Neoklis Polyzotis. *Social Networking on top of the WebdamExchange System*. In ICDE, Hannover Allemagne, 04 2011.

[Arenas 02] Marcelo Arenas, Wenfei Fan & Leonid Libkin. *Consistency of XML Specifications*. In PODS, 2002.

[Axml] Axml. *http://activexml.net*.

[Beeri 91] Catriel Beeri & Raghu Ramakrishnan. *On the power of magic*. J. Log. Program., vol. 10, no. 3-4, pages 255–299, 1991.

[Benedikt 08] Michael Benedikt, Wenfei Fan & Floris Geerts. *XPath satisfiability in the presence of DTDs*. J. ACM, vol. 55, no. 2, pages 1–79, 2008.

[Benedikt 09] Michael Benedikt & James Cheney. *Schema-Based Independence Analysis for XML Updates*. PVLDB, vol. 2, no. 1, pages 61–72, 2009.

[Benedikt 10] Michael Benedikt & James Cheney. *Destabilizers and Independence of XML Updates*. PVLDB, vol. 3, no. 1, pages 906–917, 2010.

[Bhatia 09] Gaurav Bhatia, Yupeng Fu, Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, Alin Deutsch & Yannis Papakonstantinou. *FORWARD: Design Specification Techniques for Do-It-Yourself Application Platforms*. In WebDB, 2009.

[Bhattacharya 05] K. Bhattacharya*et al. A model-driven approach to industrializing discovery processes in pharmaceutical research*. IBM Systems Journal, vol. 44, no. 1, pages 145–162, 2005.

[Bhattacharya 07a] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam & F. Y. Wu. *Artifact-centered operational modeling: Lessons from customer engagements*. IBM Systems Journal, vol. 46, no. 4, pages 703–721, 2007.

[Bhattacharya 07b] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu & J. Su. *Towards formal analysis of artifact-centric business process models*. In BPM, 2007.

[Bhattacharya 07c] K. Bhattacharya, C.E. Gerede, R. Hull, R. Liu & J. Su. *Towards formal analysis of artifactcentric business process models*. In Int. Conf. on Business Process Management, 2007.

[Björklund 08] Henrik Björklund, Wim Martens & Thomas Schwentick. *Optimizing Conjunctive Queries over Trees Using Schema Information*. In MFCS, pages 132–143, 2008.

[Björklund 09] Henrik Björklund, Wouter Gelade, Marcel Marquardt & Wim Martens. *Incremental XPath evaluation*. In ICDT '09: Proceedings of the 12th International Conference on Database Theory, pages 162–173, New York, NY, USA, 2009. ACM.

[Blakeley 86a] José A. Blakeley, Neil Coburn & Per-Åke Larson. *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates*. In VLDB'86, pages 457–466, 1986.

[Blakeley 86b] Jose A. Blakeley, Per-Ake Larson & Frank Wm Tompa. *Efficiently updating materialized views*. SIGMOD Rec., vol. 15, no. 2, pages 61–71, 1986.

[Bojanczyk 06a] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin & Claire David. *Two-Variable Logic on Words with Data*. In LICS, 2006.

[Bojanczyk 06b] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin & Claire David. *Two-Variable Logic on Words with Data*. In LICS, pages 7–16, 2006.

[Bouajjani 03] A. Bouajjani, P. Habermehl & R. Mayr. *Automatic verification of recursive procedures with one integer parameter*. Theoretical Computer Science, vol. 295, pages 85–106, 2003.

[Bouajjani 07a] A. Bouajjani, P. Habermehl, Y. Jurski & M. Sighireanu. *Rewriting systems with data*. In FCT'07, volume 4639 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.

[Bouajjani 07b] A. Bouajjani, Y. Jurski & M. Sighireanu. *A Generic Framework for Reasoning about Dynamic Networks of Infinite-State Processes*. In TACAS'07, volume 4424 of *Lecture Notes in Computer Science*, pages 690–705. Springer, 2007.

[Bouyer 02] P. Bouyer. *A Logical Characterization of Data Languages*. Information Processing Letters, vol. 84, no. 2, pages 75–85, 2002.

[Bouyer 03] P. Bouyer, A. Petit & D. Thérien. *An algebraic approach to data languages and timed languages*. Information and Computation, vol. 182, no. 2, pages 137–162, 2003.

[BPEL] BPEL. *http://bpel.xml.org/*.

[Burkart 01] O. Burkart, D. Caucal, F. Moller & B. Steffen. *Verification of infinite structures.* In Handbook of Process Algebra, pages 545–623. Elsevier Science, 2001.

[Calì 08] Andrea Calì & Davide Martinenghi. *Querying Data under Access Limitations*. In ICDE, pages 50–59, 2008.

[Calvanese 09] Diego Calvanese, Giuseppe De Giacomo, Richard Hull & Jianwen Su. *Artifact-Centric Workflow Dominance*. In ICSOC/ServiceWave, pages 130–143, 2009.

[Ceri 94] Stefano Ceri & Jennifer Widom. *Deriving Incremental Production Rules for Deductive Data.* Inf. Syst., vol. 19, no. 6, pages 467–490, 1994.

[Ceri 00] Stefano Ceri, Piero Fraternali & Aldo Bongio. *Web Modeling Language (WebML): a modeling language for designing Web sites*. Computer Networks, vol. 33, no. 1-6, pages 137–157, 2000.

[Chandra 85] Ashok K. Chandra & Moshe Y. Vardi. *The implication problem for functional and inclusion dependencies is undecidable*. SIAM journal on computing, vol. 14, no. 3, pages pp. 671–677, 1985.

[Comon 97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi. Tree automata techniques and applications. 1997. release October, 1rst 2002.

[Damaggio 11] Elio Damaggio, Alin Deutsch & Victor Vianu. *Artifact Systems with Data Dependencies and Arithmetic*. In ICDT, 2011.

[David 08] Claire David. *Complexity of Data Tree Patterns over XML Documents.* In MFCS, 2008.

[Demri 08] Stéphane Demri, Ranko Lazić & Arnaud Sangnier. *Model checking freeze LTL over one-counter automata*. In Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'08), pages 490–504, 2008.

[Demri 09] Stéphane Demri & Ranko Lazić. *LTL with the freeze quantifier and register automata*. ACM Trans. Comput. Logic, vol. 10, no. 3, pages 1–30, 2009.

[Deutsch 06] Alin Deutsch, Liying Sui, Victor Vianu & Dayou Zhou. *Verification of communicating data-driven web services.* In PODS, 2006.

[Deutsch 07] Alin Deutsch, Liying Sui & Victor Vianu. *Specification and verification of data-driven Web applications*. JCSS, vol. 73, no. 3, pages 442–474, 2007.

[Deutsch 09] Alin Deutsch, Richard Hull, Fabio Patrizi & Victor Vianu. *Automatic verification of data-centric business processes*. In ICDT, 2009.

[Diao 02] Yanlei Diao, Peter M. Fischer, Michael J. Franklin & Raymond To. *YFilter: Efficient and Scalable Filtering of XML Documents.* In ICDE, pages 341–, 2002.

[Diekert 08] Volker Diekert & Paul Gastin. *First-order definable languages*. In Jörg Flum, Erich Grädel & Thomas Wilke, editeurs, Logic and Automata: History and Perspectives, volume 2, pages 261–306. 2008.

[Dong 99] G. Dong, R. Hull, B. Kumar, J Su & G Zhou. *A framework for optimizing distributed workflow executions*. In DBLP, 1999.

[DTD] DTD. *http://www.w3.org/TR/REC-xml/#dt-doctype*.

[Emerson 90] E. Allen Emerson. *Temporal and Modal Logic*. In J. Van Leeuwen, editeur, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[Ennals 07] Robert Ennals & David Gay. *User-friendly functional programming for Web mashups*. In ICFP, pages 223–234, 2007.

[Fan 01] Wenfei Fan & Leonid Libkin. *On XML Integrity Constraints in the Presence of DTDs*. In PODS, 2001.

[Fernández 07] Mary F. Fernández, Philippe Michiels, Jérôme Siméon & Michael Stark. *XQuery Streaming à la Carte*. In ICDE, pages 256–265, 2007.

[Figueira 09] Diego Figueira. *Satisfiability of downward XPath with data equality tests*. In PODS, pages 197–206, 2009.

[Finkel 01] Alain Finkel & Philippe Schnoebelen. *Well-Structured Transition Systems Everywhere!* Theoretical Computer Science, vol. 256, no. 1-2, pages 63–92, April 2001.

[Florescu 03] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan & Geetika Agrawal. *The BEA/XQRL Streaming XQuery Processor*. In VLDB, pages 997–1008, 2003.

[Foster 08] J. Nathan Foster, Ravi Konuru, Jérôme Siméon & Lionel Villard. *An Algebraic Approach to View Maintenance for XQuery*. In PLAN-X, 2008.

[Fritz 09] Christian Fritz, Richard Hull & Jianwen Su. *Automatic construction of simple artifact-based business processes*. In ICDT, 2009.

[Genest 08] Blaise Genest, Anca Muscholl, Olivier Serre & Marc Zeitoun. *Tree Pattern Rewriting Systems*. In ATVA, pages 332–346, 2008.

[Genest 10] Blaise Genest, Anca Muscholl & Zhilin Wu. *Verifying Recursive Active Documents with Positive Data Tree Rewriting*. CoRR, vol. abs/1003.1010, 2010.

[Georgakopoulos 95] D. Georgakopoulos, M. Hornick & A. Sheth. *An overview of workflow management: From process modeling to workflow infrastructure management*. Distributed and Parallel Databases, vol. 3, pages 119–153, 1995.

[Gerede 07a] C. E. Gerede, K. Bhattacharya & J. Su. *Static Analysis of Business Artifact-centric Operational Models*. In IEEE International Conference on Service-Oriented Computing and Applications, 2007.

[Gerede 07b] C. E. Gerede & J. Su. *Specification and Verification of Artifact Behaviors in Business Process Models*. In ICSOC, 2007.

[Gottlob 02] Georg Gottlob & Christoph Koch. *Monadic Queries over Tree-Structured Data.* In LICS, pages 189–202, 2002.

[Gottlob 05] Georg Gottlob, Christoph Koch & Reinhard Pichler. *Efficient algorithms for processing XPath queries.* ACM Trans. Database Syst., vol. 30, no. 2, pages 444–491, 2005.

[Grahne 91] G. Grahne. Problem of incomplete information in relational databases. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[Green 03] Todd J. Green, Gerome Miklau, Makoto Onizuka & Dan Suciu. *Processing XML Streams with Deterministic Automata.* In ICDT, 2003.

[Gupta 93] Ashish Gupta, Inderpal Singh Mumick & V. S. Subrahmanian. *Maintaining Views Incrementally.* In SIGMOD Conference, pages 157–166, 1993.

[Gupta 95] Ashish Gupta & Inderpal Singh Mumick. *Maintenance of Materialized Views: Problems, Techniques and Applications.* IEEE Data Eng. Bull., vol. 18, no. 2, pages 3–18, 1995.

[Harel 87] David Harel. *Statecharts: A Visual Formulation for Complex Systems*. Sci. Comput. Program, vol. 8, no. 3, pages 231–274, 1987.

[Hélouët 10] Loïc Hélouët & Albert Benveniste. *Document Based Modeling of Web Services Choreographies Using Active XML.* In ICWS, pages 291–298, 2010.

[Hull] Richard Hull. Personal communication, 2009.

[Hull 99] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar & G. Zhou. *Declarative Workflows that Support Easy Modification and Dynamic Browsing*. In Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration, 1999.

[Hull 00] R. Hull, F. Llirbat, B. Kumar, G. Zhou, G. Dong & J. Su. *Optimization techniques for data-intensive decision flows*. In Proc. IEEE Intl. Conf. on Data Engineering (ICDE), pages 281–292, 2000.

[Hull 08] Richard Hull. *Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges*. In OTM Conferences (2), pages 1152–1163, 2008.

[Imielinski 82] Tomasz Imielinski & Jr. W. Lipski. *The relational model of data and cylindrical algebras*. In PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pages 170–170, New York, NY, USA, 1982. ACM.

[Imielinski 84] Tomasz Imielinski & Witold Lipski Jr. *Incomplete Information in Relational Databases*. J. ACM, vol. 31, no. 4, pages 761–791, 1984.

[Kanellakis 95] Paris C. Kanellakis, Gabriel M. Kuper & Peter Z. Revesz. *Constraint Query Languages*. J. Comput. Syst. Sci., vol. 51, no. 1, pages 26–52, 1995.

[Kapuscinski 04] Roman Kapuscinski, Rachel Q. Zhang, Paul Carbonneau, Robert Moore & Bill Reeves. *Inventory decisions in Dell's supply chain*. Interfaces, vol. 34, no. 3, pages 191–205, 2004.

[Koch 04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt & Bernhard Stegmaier. *FluXQuery: An Optimizing XQuery Processor for Streaming XML Data*. In VLDB, pages 1309–1312, 2004.

[Kumaran 08] S. Kumaran, R. Liu & F. Y. Wu. *On the duality of information-centric and activity-centric models of business processes*. In Proc. Intl. Conf. on Advanced Information Systems Engineering (CAISE), 2008.

[Kuntschke 05] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper & Angelika Reiser. *StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures.* In VLDB, 2005.

[Lazić 07] R. Lazić, Th. Newcomb, J. Ouaknine, A. Roscoe & J. Worrell. *Nets with Tokens Which Carry Data*. In ICATPN'07, volume 4546 of *Lecture Notes in Computer Science*, pages 301–320. Springer, 2007.

[Levy 93] Alon Y. Levy & Yehoshua Sagiv. *Queries Independent of Updates*. In VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases, pages 171–181, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[Levy 96] Alon Y. Levy. *Obtaining complete answers from incomplete databases*. In In Proc. of the 22nd Int. Conf. on Very Large Data Bases (VLDB'96, pages 402–412, 1996.

[Libkin 04] Leonid Libkin. Elements of finite model theory. Springer, 2004.

[Löding 09] Christof Löding & Karianto Wong. *On Nondeterministic Unranked Tree Automata with Sibling Constraints*. In FSTTCS, pages 311–322, 2009.

[Ma 08] AHai-Tao Ma, Zhong-Xiao Hao & Yan Zhu. *Checking Satisfiability of Tree Pattern Queries for Active XML Documents*. In INFOCOMP, pages 11–18, 2008.

[Marinoiu 09] Bogdan Marinoiu. *Monitoring of Distributed Applications in Peer to Peer Systems*. PhD thesis, Université Paris Sud, 2009.

[Martin 03] D. Martin*et al. OWL-S: Semantic Markup for Web Services, W3C Member Submission*, November 2003.

[McIlraith 01] S. A. McIlraith, T. C. Son & H. Zeng. *Semantic web services*. IEEE Intelligent Systems, vol. 16, no. 2, pages 46–53, 2001.

[Meier 10] Michael Meier, Michael Schmidt, Fang Wei & Georg Lausen. *Semantic query optimization in the presence of types*. In PODS, pages 111–122, 2010.

[Miklau 04] Gerome Miklau & Dan Suciu. *Containment and equivalence for a fragment of XPath*. J. ACM, vol. 51, no. 1, pages 2–45, 2004.

[Miklau 07] Gerome Miklau & Dan Suciu. *A formal analysis of information disclosure in data exchange*. J. Comput. Syst. Sci., vol. 73, no. 3, pages 507–534, 2007.

[Milner 89] R. Milner. Communication and concurrency. Prentice-Hall, Inc., 1989.

[Mok 02] W.Y. Mok & D.P. Paper. *Using Harel's Statecharts to Model Business Workflows*. J. of Database Management, vol. 13, no. 3, pages 17–34, 2002.

[Motwani 03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein & R. Varma. *Query Processing, Approximation, and Resource Management in a Data Stream Management System.* In CIDR, 2003.

[Muscholl 04] Anca Muscholl, Thomas Schwentick & Luc Segoufin. *Active Context-Free Games*. In STACS, pages 452–464, 2004.

[Narayanan 02] S. Narayanan & S. McIlraith. *Simulation, Verification and Automated Composition of Web Services*. In WWW, 2002.

[Neven 04] F. Neven, T. Schwentick & V. Vianu. *Finite State Machines for Strings Over Infinite Alphabets*. ACM Transactions on Computational Logic, vol. 5, no. 3, pages 403–435, 2004.

[Nigam 03a] A. Nigam & N.S. Caswell. *Business artifacts: An approach to operational specification*. IBM Systems Journal, vol. 42, no. 3, pages 428–445, 2003.

[Nigam 03b] Anil Nigam & Nathan S. Caswell. *Business artifacts: An approach to operational specification*. IBM Systems Journal, vol. 42, no. 3, pages 428–445, 2003.

[Onizuka 05] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami & Takashi Honishi. *Incremental maintenance for materialized XPath/XSLT views.* In WWW, pages 671–681, 2005.

[O'Reilly] Tim O'Reilly. *What is Web 2.0?, http://www.oreilly.com/web2/archive/what-is-web-20.html*.

[Ronen 07] Royi Ronen & Oded Shmueli. *Evaluation of datalog extended with an XPath predicate*. In WIDM, pages 9–16, 2007.

[Sawires 05] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal & K. Selçuk Candan. *Incremental Maintenance of Path Expression Views.* In SIGMOD Conference, pages 443–454, 2005.

[Segoufin 07] Luc Segoufin. *Static analysis of XML processing with data values*. SIGMOD Record, vol. 36, no. 1, pages 31–38, 2007.

[Seidl 03] Helmut Seidl, Thomas Schwentick & Anca Muscholl. *Numerical document queries*. In PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 155–166, New York, NY, USA, 2003. ACM.

[Spielmann] M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. Ph.D. thesis, RWTH Aachen, 2000.

[Spielmann 03] M. Spielmann. *Verification of Relational Transducers for Electronic Commerce*. JCSS., vol. 66, no. 1, pages 40–65, 2003. Extended abstract in PODS 2000.

[Stegmaier 04] Bernhard Stegmaier, Richard Kuntscke & Alfons Kemper. *StreamGlobe: adaptive query processing and optimization in streaming P2P environments*. In ACM International Conference Proceeding Series; Vol. 72, 2004.

[van Benthem 76] Johan van Benthem. *Modal correspondence theory*. PhD thesis, Mathematish Instituut & Instituut voor Grondslagenonderzoek, Univ. of Amsterdam, 1976.

[van der Aalst 98] Wil M. P. van der Aalst. *The Application of Petri Nets to Workflow Management*. Journal of Circuits, Systems, and Computers, vol. 8, no. 1, pages 21–66, 1998.

[van der Aalst 02] W. M. P. van der Aalst & A. H. M. ter Hofstede. *Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages.* In Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, 2002, 2002.

[van der Aalst 04] W.M.P. van der Aalst. *Business process management demystified: A tutorial on models, systems and standards for workflow management*, 2004. In *Lectures on Concurrency and Petri Nets*.

[Vardi 96] Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*. In Logics for Concurrency: Structure versus Automata, volume 1043 of LNCS, pages 238–266. Springer-Verlag, 1996.

[Vieille 89] L. Vieille. *Recursive query processing: the power of logic*. Theor. Comput. Sci., vol. 69, no. 1, pages 1–53, 1989.

[Wang 05] J. Wang & A. Kumar. *A Framework for Document-Driven Workflow Systems*. In Business Process Management, pages 285–301, 2005.

[Wikipedia] Wikipedia. *http://en.wikipedia.org/wiki/Casting_%28performing_arts%29*.

[Wong 07] Karianto Wong & Christof Löding. *Unranked Tree Automata with Sibling Equalities and Disequalities*. In ICALP, pages 875–887, 2007.

[WSDL] WSDL. *http://www.w3.org/TR/wsdl*.

[Yahoo Pipe] Yahoo Pipe. *pipes.yahoo.com*.

[Zhao 09] Xiangpeng Zhao, Jianwen Su, Hongli Yang & Zongyan Qiu. *Enforcing Constraints on Life Cycles of Business Artifacts*. In TASE, pages 111–118, 2009.

Part IV.

Appendix

Appendix A.

Satisfiability and Relevance: Proofs

A.1. Notations

In the main text, a generalized tuple $(u, \mathcal{C})$ is a tuple of variables and the constraints are equality constraints over variables and labels. In this section and in all the proofs, we use a definition that is more convenient for the proofs. It is a pair consisting of (i) a tuple of variables and labels and equality constraints over variables. The two definitions are clearly equivalent and one can transform from one to the other in PTIME.

Let q be a query, t an active document, p, n some nodes, respectively in q, t; and u a tuple over all the variables of q. Let $\pi_p(u)$ be the projection of u over the variables appearing in the subtree rooted at p. The query $q(u)$ is obtained from q by replacing the variable $\$i$ by $u(\$i)$. This definition is extended to all tuples over a subset of the variables appearing in q. In the rest of the paper, we suppose that there are no two nodes belonging to π labeled by the same variable. Then:

- $desc_q(p)$ and $desc_t(n)$ are the set of descendants of p for the query q and the set of descendants of n for the tree t.
- $\lfloor p \rfloor_q$ and $\lfloor n \rfloor_t$ are the subtrees rooted at p and n, respectively.
- $//q$ is the query with a root that (i) is labeled $*$; (ii) has q as a single subtree ; and (iii) the edge from the root is in $E_{//}$.

Finally, $[p]_u$ is the query defined as follows:

- if the edge leading to p is in $E_/$, then $[p]_q = \lfloor p \rfloor_q$ and
- if the edge leading to p is in $E_{//}$, then $[p]_q = \lfloor p \rfloor_q \vee //\lfloor p \rfloor_q$.

Note that $[p]_u$ is not a tree pattern query but a union of such queries.

Figure A.1 shows a query q and the derived queries : $\lfloor c \rfloor_q$, $//\lfloor c \rfloor_q$ and $[c]_q$. When q, t are understood, we use $desc(p)$, $\lfloor p \rfloor$ and $\lfloor n \rfloor$, $[p]$.

A.2. Satisfiability

A.2.1. Proof of Theorem 2.6

We use Algorithm A.1 to construct datalog programs that compute satisfiability in PTIME in the size of the documents. We explain some aspects of the datalog programs and give a proof of correctness. In the program, we denote by $Activevar(p)$ the set of variables that appear in $\lfloor p \rfloor$ and that are needed in the output or that need to be carried on for checking a join constraint. That set can be computed in a preprocessing phase, ignoring the data. For each node p occurring in the query, we consider

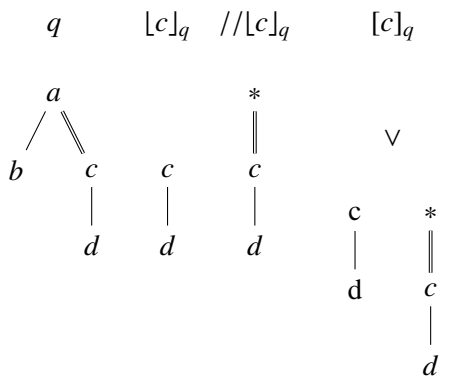


Figure A.1.: Examples of queries

a predicate called $\widehat{p}$. The arity of $\widehat{p}$ is the number of active variables plus one. Its first column, with attribute name *node*, is used for the node identifiers and each other column corresponds to a variable in $Activevar(p)$. We denote by $C(p)$ the equality constraints associated to the relation p. Observe how $C(p)$ is used in Algorithm A.1. The constraint $C(p)$ in the rule for $\widehat{p}$ contains :

- $u_{p'}(\$1) = u_{p''}(\$1)$ if $p', p'' \in children(p) \cup \{p\}$ and $\$1 \in Activevar(p') \cap Activevar(p'')$.
- $u_p(\$1) = x$ where $label(x, n)$ occurs in the rule of $\widehat{p}$ and $\lambda(p) = \$1 \in Activevar(p)$.
- $u_{p'}(\$1) = x$ where $label(x, n)$ occurs in the rule of $\widehat{p}$ and $\lambda(p') = \$1 \in Activevar(p')$.

The constraint $C(p)$ in the rule for $\widehat{p}$ contains nothing else.

Algorithm A.1 Satisfiability for queries

Data: a tree-pattern query q
Result: the datalog program $\widehat{P}_q$
begin
  for $p = root(q)$ **do**
    $\widehat{P}$ +:= $\widehat{q}() \leftarrow \widehat{p}(u, n), root(n)$
  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**
    if *p labeled by* $b \in \mathcal{L}$ **then**
      δ +:= $\widehat{p}(n, u) \leftarrow label(b, n), \widehat{p}_{i_1}(n_1, u_{i_1}), \ldots, \widehat{p}_{j_1}(n'_1, u_{j_1}), \ldots,$
      $child(n, n_1), ..., descendant(n, n'_1), \ldots, C(p)$ where $(p, p_{i_x}) \in E_/, (p, p_{j_y}) \in E_{//},$
    else
      δ +:= $\widehat{p}(n, u) \leftarrow label(x, n), \widehat{p}_{i_1}(n_1, u_{i_1}), \ldots, \widehat{p}_{j_1}(n'_1, u_{j_1}), \ldots,$
      $child(n, n_1), ..., descendant(n, n'_1), \ldots, C(p)$ where $(p, p_{i_x}) \in E_/, (p, p_{j_y}) \in E_{//},$
  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**
    δ +:= $\widehat{p}(n, u) \leftarrow function(n)$ where for each $\$i \in Activevar(p), u(\$i) = x_i$
end

The next lemma states the soundness of the datalog program.

Lemma A.1. *Let q, $\widehat{P_q}$, I, be a query, the datalog program associated to q, an active document. Let be n and $(u, \mathcal{C})$ be a data node of I and a generalized tuple. Let p be a node of q. If $(n, u, \mathcal{C}) \in \widehat{p}$ in $\widehat{P_q}(I)$, then $(u, \mathcal{C}) \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$.*

Proof. The proof is by induction.

- Let p be a leaf of q.
  - $Activevar(p) = \{\$i\}$. Then p is labeled by a variable $\$i$. So, in compliance with Algorithm A.1, $(n, x,) \in \widehat{p}$ iff n is a function node, and $(n, a) \in \widehat{p}$ iff n is a data node labeled by a. So for the second case, it implies that $\lfloor n \rfloor \models \diamond\lfloor p \rfloor(a)$.
  - $Activevar(p) = \varnothing$. According to Algorithm A.1, $(n) \in \widehat{p}$ iff n is a function node or p is labeled by a variable or p and n are labeled by the same label a. In the two cases, it implies that $\lfloor n \rfloor \models \diamond\lfloor p \rfloor$.
- Let p be a query node and p_{i_y} one of its children such that $(p, p_{i_y}) \in E_/$ (and similarly for $E_{//}$). Let $(u, \mathcal{C})$ be a generalized tuple and n be a data node such that $(n, u, \mathcal{C}) \in \widehat{p}$. Let $\{(n_x, u_x, \mathcal{C}_x)\}$ be a set of generalized tuples belonging to the relations $\widehat{p_{i_y}}$ such that they imply that (n, u) belongs to $\widehat{p}$ following the rules described in Algorithm A.1. We remark that the generalized tuples $(n_x, u_x, \mathcal{C}_x)$ have solvable constraints. Let φ a valuation of the generalized tuple u. The restriction of φ over common attributes of u and u_x is a partial valuation of u_x which can be extended to a complete valuation of u_x. This valuation is denoted by φ_x. We build a sequence of updates ω such that $\omega(\lfloor n \rfloor) \models \lfloor p \rfloor(\varphi(u))$.
  - If n_x is a function node. Then, the call Id associated can receive a tree satisfying $[p_x](\varphi_x(u_x))$.
  - If n_x is a data node. By induction, there is a sequence ω_x such that $\omega_x(\lfloor n_x \rfloor) \models \lfloor p_x \rfloor(\varphi_x(u_x))$.

  Then, by aggregating all the sequences of updates in the global sequence of updates ω, we have that $\omega(\lfloor n \rfloor) \models \lfloor p \rfloor(\varphi(u))$. So, for each instantiation φ of u, there is a sequence ω such that $\omega(\lfloor n \rfloor) \models \lfloor p \rfloor(\varphi(u))$.

Therefore $(u, \mathcal{C}) \diamond \lfloor p \rfloor(\lfloor n \rfloor)$. □

The next lemma states the completeness of the datalog program.

Lemma A.2. *Let q, $\widehat{P_q}$, I, $(u, \mathcal{C})$ be a query, the datalog program associated to q, an active document and a generalized tuple. Let p and n be a node of q and a data node of I. If $(u, \mathcal{C}) \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$ then there exists a generalized tuple $(n, u', \mathcal{C}')$ belonging to $\widehat{P_q}(I)$ such that $(u, \mathcal{C}) \sqsubseteq (u', \mathcal{C}')$.*

Proof. The proof is by induction.

- Let p, $(u, \mathcal{C})$ and n be a leaf of q, a generalized tuple and a data node of d such that $(u, \mathcal{C}) \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$. The node n is labeled by b. Because n is a data node, u has its value in $\mathcal{L}$, and $u = b$. Remark that the body of $\widehat{p}$ is equal to $label(b, n)$. Then, $(n, u, \mathcal{C})$ is in $\widehat{p}$.
- Let p be a query node and p_{i_x} one of its children such that $(p, p_{i_x}) \in E_/$. Let $(u, \mathcal{C})$ and n be a tuple and a data node such that $(u, \mathcal{C}) \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$. Let φ be a valuation of $(u, \mathcal{C})$ such that each variable of u is associated to some fresh label when it is possible. So, there exists a sequence ω of updates such that $\omega(\lfloor n \rfloor) \models \lfloor p \rfloor(\varphi(u))$. Thus, there is a valuation ν from

$\lfloor p \rfloor(\varphi(u))$ to $\omega(\lfloor n \rfloor)$. For each p_x child of p, we can define a generalized tuple $(u_x, \mathcal{C}_x)$ built from $\varphi(u)$ by projecting it on variables common to $Activevar(p)$ and $Activevar(p_x)$ and then by extending it to other variables of $Activevar(p_x)$ according to the valuation ν. So $(u_x, \mathcal{C}_x)$ belongs to $\omega(\lfloor n_x \rfloor)$. We have to consider two cases :

- n_x belongs to $\lfloor n \rfloor$. By induction, there is a generalized tuple $(u'_x, \mathcal{C}'_x)$ such that $(n_x, u'_x, \mathcal{C}'_x) \in \widehat{p_x}$ and $(u_x, \mathcal{C}_x) \sqsubseteq (u'_x, \mathcal{C}'_x)$.
- n_x does not belong to $\lfloor n \rfloor$. So there is a function node n'_x which is a sibling of an ancestor of n_x. In this case, we have $(n'_x, u'_x, \varnothing) \in \widehat{p_x}$. The tuple $(u'_x,)$ is composed by variables without constraints, so $(u_x, \mathcal{C}_x) \sqsubseteq (u', \varnothing)$.

By using the rule of $\widehat{p}$ with the generalized tuples $(n_x, u_x, \mathcal{C}_x)$ or $(n'_x, u'_x, \mathcal{C}'_x)$, we can define a generalized tuple $(u', \mathcal{C}')$ such that $(n, u', \mathcal{C}') \in \widehat{p}$. Because the tuples in $\widehat{P}$ take values only in the labels of q and I and in the variables, the choice of the valuation φ at the beginning does not change the definition of u'_x (see Lemma A.3 further) . Finally, the rule is monotone and for each $x, (u_x, \mathcal{C}_x) \sqsubseteq (u'_x, \mathcal{C}'_x)$, so $(u, \mathcal{C}) \sqsubseteq (u', \mathcal{C})$.

□

This concludes the proof of Theorem 2.6.

A.2.2. Proof of Theorem 2.7

To show NP-HARDNESS, we use a reduction of the evaluation problem for tree-pattern queries that is known to be NP-COMPLETE. The NP-COMPLETENESS of the evaluation of our tree-pattern queries can be shown by adapting either the proof of Theorem 7.3 of [Gottlob 02] or the proof of the satisfaction problem for injective tree-patterns in [David 08].

We now prove that the satisfiability problem is in NP. Let q be Boolean query and I an active document. To show that $I \models \diamond q$, it suffices to exhibit a sequence ω of insertions and a valuation ν of q in $\omega(I)$. First, observe that if such a sequence exists, there is one with a number of insertions bounded by $|q|$ and the size of inserted trees also bounded by $|q|$ (see Lemma A.4). Furthermore, observe that we need only to consider a polynomial number of labels (we have to guess values for variables. See Lemma A.3). Then we have to check (in polynomial time) that the given *candidate valuation* is successful. The query to evaluate over $\omega(I)$ is a no-join tree-pattern query (the variables are replaced by the values that have been guessed). Since the satisfaction of a no-join tree-pattern query is polynomial in the sizes of the query and the tree [Gottlob 02, Miklau 04], membership in NP follows. We now prove the two auxiliary results that are used in the proof of the theorem, namely Lemma A.3 and Lemma A.4.

Let q, I, $(u, \mathcal{C})$ be a query, an active document, and a generalized tuple. By definition, $I \models \diamond q((u, \mathcal{C}))$ iff for all instantiations θ of $(u, \mathcal{C})$, $I \models \diamond q(\theta(u))$. The following lemma shows that it suffices to check one particular instantiation.

Lemma A.3. *Let q, I, $(u, \mathcal{C})$ be a query, an active document and a generalized tuple. Let θ be a particular instantiation of u that associates to each $\$i$, a distinct, fresh label, i.e. a label not appearing in I or q. Then $(u, \mathcal{C}) \in \diamond q(I)$ iff $I \models \diamond q(\theta(u))$.*

Proof. ($\Rightarrow$) If $(u, \mathcal{C}) \in \diamond q(I)$, for each instantiation of $(u, \mathcal{C})$, θ', $I \models \diamond q(\theta'(u))$. Thus, in particular, $I \models \diamond q(\theta(u))$.

($\Leftarrow$) Now suppose that $I \models \diamond q(\theta(u))$ for this particular θ. Let ω be an update sequence so that $\omega(I) \models q(\theta(u))$. Let θ' be an arbitrary instantiation. By construction of θ, there exists a call Id f

such that for all $\$i$, $\theta'(\$i) = f(\theta(\$i))$. Let ω' be the update sequence obtained from ω by replacing each constant $\theta(\$i)$ by $\theta'(\$i) = f(\theta(\$i))$. Clearly, $\omega'(I) \models q(\theta'(u))$, so $I \models \Diamond q(\theta'(u))$. Since θ' is arbitrary, $(u, \mathcal{C}) \in \Diamond q(I)$. □

We now show that for a tuple u, u is satisfiable iff there exists a sequence of updates ω such that (i) it length is bounded by $|q|$ and (ii) each subtree K of an update belonging to ω has its size bounded by the size of q. For that, we use the definition of scenarios mentioned in Section 2.6.

Algorithm A.2 computes a set of scenarios denoted $Scen(q, I)$.

This program takes an active I document and builds scenarios for I and q. It is derived from Algorithm A.1. We assume some orders for the nodes of the query.

Algorithm A.2 Scenario for queries

Data: a tree-pattern query q
Result: the datalog program $relevant_q$ such that $relevant_q(I) = Scen(q, I)$
begin
  for $p = root(q)$ **do**
    δ +:= $relevant_q(u, \mathcal{P}) \leftarrow relevant_p(n, u, \mathcal{P}), root(n)$
  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**
    if p *labeled by* $b \in \mathcal{L}$ **then**
      δ +:= $relevant_p(n, u, \mathcal{P}) \leftarrow label(b, n), relevant_{p_{i_1}}(n_1, u_{i_1}, \mathcal{P}_{i_1}), \ldots, relevant_{p_{j_1}}(n'_1, u_{j_1}, \mathcal{P}_{j_1}), \ldots,$
      $child(n, n_1), descendant(n, n'_1), \ldots, C(p)$ where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
      $\Pi_{descp_{i_x}}(\mathcal{P}) = \mathcal{P}_{i_x}, \Pi_{descp_{i_x}}(\mathcal{P}) = \mathcal{P}j_y, \mathcal{P}(p) = \star$
    else
      δ +:= $relevant_p(n, u, \mathcal{P}) \leftarrow label(x, n), relevant_{p_{i_1}}(n_1, u_{i_1}, \mathcal{P}_{i_1}), \ldots, relevant_{p_{j_1}}(n'_1, u_{j_1}, \mathcal{P}_{j_1}), \ldots,$
      $child(n, n_1), descendant(n, n'_1), \ldots, C(p)$ where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
      $\Pi_{descp_{i_x}}(\mathcal{P}) = \mathcal{P}_{i_x}, \Pi_{descp_{i_x}}(\mathcal{P}) = \mathcal{P}j_y, \mathcal{P}(p) = \star$
  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**
    δ +:= $relevant_p(n, u, \mathcal{P}) \leftarrow function(n)$ where $\forall \$i \in Activevar(p), u(\$i) = x_i$
    $\mathcal{P}(p) = \lambda(n), \forall n' \in desc(p), \mathcal{P}(p') = \bullet$
end

Lemma A.4 gives the correctness and the soundness of this algorithm for extracting the information needed for proving document satisfiability.

Lemma A.4. *Let q, I be a query and an active document. Let u and ω be a tuple over q and a sequence of updates. Then ω(I) satisfies the Boolean query q(u) iff there exists a scenario* $(u', \mathcal{C}, \mathcal{P})$ *in* Scen(q, I) *such that* $(u, \varnothing) \sqsubseteq (u', \mathcal{C})$ *and ω implements* $(u, \varnothing, \mathcal{P})$.

Proof. Observe that the graph of dependencies of the relations in the program constructed by Algorithm A.2 is a tree. The proof is by induction on the graph of dependencies of the relations. Let n, p, u, ω be a data node of I, a node of q, a tuple over the variables of $[p]$, and a sequence of updates. We show that

$\omega(\lfloor n \rfloor)$ satisfies $\lfloor p \rfloor(u)$ iff there is a scenario $(u', \mathcal{C}, \mathcal{P}) \in relevant_p$ such that $(u, \varnothing) \sqsubseteq (u', \mathcal{C})$ and ω implements $(u', \mathcal{C}, \mathcal{P})$.

The different parts of the inductions are straightforward. This proves the lemma. □

This concludes the proof of Theorem 2.7.

A.2.3. Systems of active documents

We now consider systems of active documents where a call Id may correspond to a subscription to a query over some active document of the system. This introduces recursion in the evaluation.

A.2.3.1. Proof of Theorem 2.8

Let $\mathcal{S}$ and d be an axlog system with empty queues and a document belonging to this system. First, we can forget the order of applications of the updates in the system since the system is clearly confluent. The program datalog $\widehat{P_q}$ is adapted as follows:

1. There are new extended relations $d(n)$ (the node n belongs to the document d), $\xi_{q,t_o}(n)$ ($\xi(\lambda(n)) = (q, t_o)$)
2. The relations $\bar{p}_{q_i}(u, u')$ denote the result tuple of querying a tree $t_o(u')$.
3. A program $\tilde{P}_{(d,(q,t_o))}$ is built from the program $\tilde{P}_q$ by adding the conditions $d(n)$ in each rule. The rules $\widehat{p}(n, u) \leftarrow fun(n)$ are replaced by rules

$$\tilde{p}(n, u) \leftarrow fun_{(q_i,t_o)}(n), d(n), \bar{p}_{q_i}(u, u'), \tilde{q}_i(u').$$

By applying naive evaluation of the datalog program iteratively, we can demonstrate that $\Delta_{j+1} = Q^{j+1}(I) - Q^j(I)$ is the set of updates resulting of applying the updates Δ_j in the system, the external updates are added after the first iteration. So, because of confluence, each tuple computed by the program is indeed a satisfiable tuple.

This program shows that the problem is solvable in PTIME in the size of the document and in EXPTIME in the size of the system (size of the queries and size of the document).

The EXPTIME-HARDNESS is by reduction of the evaluation of datalog program that is known to be EXPTIME-HARD. That ends the proof.

A.3. Typed documents

In this section, we provide the proofs concerning satisfiability for documents constrained by schemas based on DTDs. Reasoning over DTDs and reduced trees is more difficult than over DTDs with classical trees. In this context, we propose a specific kind of DTDs, called *reduced* DTDs, easier to manipulate. First, the definition of reduced DTDs is given. We introduce an algorithm to transform a DTD into an equivalent reduced DTD. Then, we extend this notion to Axlog schemas. An algorithm solving the satisfiability problem for reduced DTDs is given. Finally, the study of the complexity of this algorithm provides the proofs of Theorems 2.11 and 2.12.

A.3.1. Reduced DTDs

Reasoning about DTDs and reduced trees is not obvious as shown in Section 2.4.

The goal of this subsection is to propose a definition of DTD called *reduced DTD* such that the problems exhibited by the example in Section 2.4do not appear. Moreover, we explain how to find from a DTD an equivalent reduced DTD.

Let F be a forest. We consider the multiset of the roots of the trees in F. We denote by $root\{\mathcal{F}\}$ the multiset of labels obtained by replacing in this set, each call to a function w by w and each label in $\mathcal{L} - \Sigma$ by dom. For a unordered-DTD (τ, Σ, W, r), we denote by the $\lambda(\tau)$ the set equals to $\Sigma \cup \{dom\} \cup W$.

Definition A.5. An unordered-DTD (Σ, W, d, τ) is *reduced* iff for each $a \in \Sigma$, for each multiset M, $M \vdash \tau(a)$, there is a reduced forest F of τ such that (i) each tree of the forest satisfies the children constraints* τ and (ii) $root\{F\} = M$.

We show that each unordered-DTD has a reduced DTD that defines the same set of reduced trees. But first, we consider the notion of disjunctive DTD that is useful towards the notion of reduced DTD, we study. We show that for each unordered-DTD there exists an equivalent one that is in a disjunctive form and from each disjunctive DTD, there exists a reduced DTD that defines the same set of reduced trees.

A.3.1.1. Disjunctive DTD

Definition A.6. Let (τ, Σ, W, r) be an unordered DTD. It is a *disjunctive DTD* iff there exists an integer denoted by $max(\tau)$ such that for each $a \in \Sigma$, $\tau(a) = \bigvee \varphi_i$ where for each i,

(+) φ_i is a conjunction of atoms of the form $|b| = k$, $k \leqslant max(\tau)$ or $|b| > max(\tau)$.

(++) for each $b \in \Sigma \cup \{dom\} \cup W$, a term $|b|$ op k appears exactly once in φ_i (possibly the term $|b| = 0$)

(+++) for each j, $i \neq j$, the formula $\varphi_i \wedge \varphi_j$ is unsatisfiable.

The following lemma shows that for each unordered-DTD, there is an equivalent disjunctive DTD.

Lemma A.7. *Let $D = (\tau, \Sigma, W, r)$ be an unordered-DTD. There exists a disjunctive DTD $D' = (\tau', \Sigma, W, r)$ such that for each $a \in \Sigma$, M a multiset of $\lambda(\tau)$, M satisfies $\tau(a)$ iff M satisfies $\tau'(a)$.*

Proof. We show how to construct a disjunctive DTD (τ', Σ, W, r) such that the sets of trees defined by τ and τ' are equal. We denote by $max(\tau)$ the largest integer appearing in the constraints of τ. The negations can be pushed to the bottom of the tree-formula. Then $\neg(|b| \text{ op } k)$ can be replaced by $|b|$ op $'k$, e.g. $\neg(|b| = 5)$ by $|b| \neq 5$. One can eliminate the undesired comparators as follows:

1. $|b| \neq k$ by $(|b| < k) \vee (|b| > k)$
2. $|b| \leqslant k$ by $(|b| < k) \vee (|b| = k)$ and $|b| \geqslant k$ by $(|b| > k) \vee (|b| = k)$
3. $|b| < k$ by $\bigvee_{i<k} |b| = i$
4. $|b| > k$ by $\bigvee_{k<i\geqslant max(\tau)} |b| = i \vee |b| > max(\tau)$

After applying these substitutions, each constraint $\tau(a)$ can be rewritten into a disjunctive formula $\bigvee \varphi_i$ where each φ_i satisfies Properties (+) and (++). For each φ_i and each b, if a term $|b|$ op k does not appear in φ then the term $|b| = 0$ is added to φ_i. Moreover, if there are two terms of the form $|b|$ op k in a formula φ_i, the formula is rewritten as follows:

There are two terms $|b| = k_1$ and $|b| = k_2$. If k_1 and k_2 are equal then one of the terms is removed. Otherwise, the formula φ_i is removed.

There are a term $|b| = k$ and a term $|b| > max(\tau)$. The formula φ_i is removed.

*The children constraints are the constraints over children of a node imposed by τ, see Section 2.4.

There are two terms $|b| > max(\tau)$ and $|b| > max(\tau)$. Then one of the terms is removed.

Then, each formula φ_i satisfies Property (++). Remark that for each a, for each $i, j, i \neq j$, $\varphi_i, \varphi_j \in \tau(a)$, $\varphi_i \wedge \varphi_j$ is satisfiable iff φ_i is equal to φ_j. Then for each formula $\tau(a)$, it is sufficient to keep a unique representative of each φ_i of $\tau(a)$. After this last operation, we obtain τ' with the desired property.

So, (τ', Σ, W, r) is a disjunctive DTD and for each $a \in \Sigma$, M a multiset of Σ, $M \vdash \tau(a)$ iff $M \vdash \tau'(a)$. □

Remark that Lemma A.7 implies that the unordered-DTD D' defines the same set of trees than D.

Observe that the transformation of a unordered-DTD D into a disjunctive DTD D may imply the explosion of the size of the children constraints of D' in the size of the children constraints τ.

A.3.1.2. From disjunctive DTD to reduced DTD

First, we show for a given disjunctive DTD (τ, Σ, W, r), the existence of a reduced DTD defining the same the set of reduced trees. This algorithm uses the auxiliary notion of number of reduced trees rooted by a label a and satisfying the children constraints τ, for each label $a \in \lambda(\tau)$. First, we assume that these numbers are given. Then, we explain how to compute them.

Definition A.8. Let (τ, Σ, W, r) and a be an unordered-DTD and a label belonging to $\lambda(\tau)$. We denote by $[a]_\tau$, the number of different reduced trees rooted by a and that satisfy the children constraints τ.

When τ is understood, we use $[a]$.

Definition A.9. Let $D = (\tau, \Sigma, W, r)$ be a disjunctive DTD. We denote by $reduced(D) = (\tau', \Sigma, W, r)$, the DTD where the children constraints of τ', are obtained by rewriting each conjunctive formula φ of the constraint $\tau(a)$, for each $a \in \Sigma$ as follows:

1. If for each term $|b| = k$ of φ, $[b]$ is greater or equal than k and for each term $|b| > max(\tau) \in \varphi$, $[b] = \infty$ then φ is kept as it is.

2. If for each term $|b| = k$ of φ, $[b]$ is greater or equal than k, for each term $|c| > max(\tau)$ of φ, $[c] > max(\tau)$ and there exists a term $|d| > max(\tau)$ of φ such that $[d] \neq \infty$ then φ is rewritten as follows:
   - Each term $|b| = k$ is kept.
   - Each term $|c| > max(\tau)$ such that $[c] = \infty$ then the term is kept.
   - Finally, each term $|d| > max(\tau)$ such that $[d] < \infty$ then the term is replaced by $|d| > max(\tau) \wedge |d| \leqslant [d]$.

3. In the others cases, the formula φ is removed from the disjunction.

Remark that the construction does not give a disjunctive form (because of the second rule). By applying Lemma A.7, we can turn it into disjunctive form without losing the property that the resulting DTD is reduced.

Example A.10. The next unordered-DTD $D = (\tau, \{a, b\}, \varnothing, a)$ is not a reduced DTD.

$$\begin{array}{lll} a & \rightarrow & |b| > 0 \wedge |a| = 0 \wedge |dom| = 0 \\ b & \rightarrow & |a| = 0 \wedge |b| = 0 \wedge |dom| = 0 \end{array}$$

And the following unordered-DTD is $reduced(D)$= $(\tau', \{a, b\}, \varnothing, a)$ built from the previous unordered-DTD.

$$\begin{array}{lll} a & \rightarrow & |b| = 1 \wedge |a| = 0 \wedge |dom| = 0 \\ b & \rightarrow & |a| = 0 \wedge |b| = 0 \wedge |dom| = 0 \end{array}$$

The next theorem shows the correctness of the construction of $reduced(D)$.

Theorem A.11. *Let $D = (\tau, \Sigma, W, r)$ be a disjunctive DTD. Then, the unordered-DTD* reduced*(D) is a reduced DTD and the set of reduced trees satisfying D is the same as the one satisfying* reduced*(D).*

Proof. By construction of the constraints of $reduced(D) = (\tau', \Sigma, W, r)$, we can remark that for each conjunctive formula φ of $\tau'(a)$, there is a formula φ' in $\tau(a)$, such that each multiset satisfying φ satisfies φ'. Then, the trees satisfying $reduced(D)$ satisfy D. So the set of reduced trees satisfying $reduced(D)$ is included in the set of reduced trees satisfying D.

Suppose by contradiction, that there is a reduced tree t satisfying D and not $reduced(D)$. Let n be a node of t labeled by a such that its children do not satisfy $\tau'(a)$. By construction of τ', it implies that there exists a label b in $\lambda(\tau)$ such that the number of children of n labeled b is greather than $[b]$. But the subtrees of n labeled by b form a reduced forest. So, there is a contradiction. So, the set of reduced trees satisfying τ is included in the set of reduced trees satisfying τ'. Thus, the set of reduced trees satisfying τ is equal to the set of reduced trees satisfying τ'.

The fact that $reduced(D)$ is reduced comes from its construction. For each a, for each conjunctive formula φ of $\tau'(a)$, for each term $|b| = k$ of φ, there exists a reduced forest of trees rooted by b and satisfying the children constraints of τ', because $k \leqslant [b]$. For each term of φ, $|c| > max(\tau)$, there exists an infinite set of reduced trees rooted by c and satisfying the children constraints τ'. So, for any multiset M satisfying φ, there is a reduced forest F such that any tree of F satisfies the children constraints τ' and $root\{F\} = M$. □

Computation of the numbers of reduced trees Let $D = (\tau, \Sigma, W, r)$ be a disjunctive DTD. We provide an algorithm, namely Count-RT (for count reduced trees) which computes for each a in $\lambda(\tau)$, $[a]_\tau$. It is presented in Figure A.3. Algorithm Count-RT uses the following symbols: for each $a \in \lambda(\tau)$ the number Count-RT$(D)(a)$, the function τ_{Active} and the sets N_{Active} and E_{Active}. For each $a \in \Sigma$, the subset $\tau_{Active}(a)$ of conjunctive formulas is a subset of $\tau(a)$. A conjunctive formula φ appears in $\tau_{Active}(a)$ iff there exists a reduced tree rooted at a such that the multiset M of labels of the children of its root satisfies φ. The set N_{Active} is a set of labels rooting reduced trees satisfying the children constraints of τ, i.e $a \in N_{Active}$ iff Count-RT$(D)(a) > 0$. The set of edges E_{Active} represents the known dependencies of a label a to another b. The pair (a, b) belongs to E_{Active} iff there exists a conjunction $\varphi \in \tau_{Active}(b)$ such that the term $|a|$ op k appears in φ and is different from the term $a = 0$. At the end of the computation of Count-RT, for each $a \in \lambda(\tau)$, Count-RT$(D)(a)$ should be equal to the number $[a]_\tau$. Lemma A.14 shows this property.

Then, we define Count-RT$(D)(a)$ for $a \in \Sigma$ as follows:

1. if a belongs to a cycle in (N_{Active}, E_{Active}) then Count-RT$(D)(a) = \infty$
2. if a does not belongs to a cycle then Count-RT$(D)(a) = \sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b \in \lambda(\tau)} Count(b, \varphi_i)$

where $Count(a, \varphi)$ is defined as follows:

- For $(|a| = k) \in \varphi$, $Count(a, \varphi) = \binom{\text{Count}-\text{RT}(D)(a)}{k}$
- For $(|a| > max(\tau)) \in \varphi$, $Count(a, \varphi) = \sum_{i=max(\tau)+1}^{\text{Count}-\text{RT}(D)(a)} \binom{\text{Count}-\text{RT}(D)(a)}{i}$

The numbers Count-RT$(D)(a)$ and $Count(a, \varphi)$ take their values in $\mathbb{N} \cup \{\infty\}$. The following algebra over $\mathbb{N} \cup \{\infty\}$ is used: $k * \infty = \infty$ for $k \neq 0$; $k + \infty = \infty$; $\sum_{i=k}^{\infty} u_i = \infty, u_i \in \mathbb{N}^+ \cup \{\infty\}$. We assume that $\infty * 0 = 0$; for $k \neq 0$, $\binom{\infty}{k} = \infty$ and for $k = 0$, $\binom{\infty}{k} = 1$. We assume also that $\sum_{a \in A} f(a) = 0$ if $A = \emptyset$.

Algorithm Count-RT is initialized as follows: for each $a \in \Sigma$, Count-RT$(D)(a)$= 0; for each $w \in W$, Count-RT$(D)(w)$= ∞, Count-RT(D)(dom)= ∞, N_{Active}, E_{Active}, $\tau_{Active}(a)$, for each $a \in \Sigma$ are equal to the empty set. The algorithm has two embedded loops called C-graph and C-value. Intuitively, the (C-value) loop computes the values Count-RT$(D)(a)$ for the DTD limited to the labels appearing in N_{Active} and the children constraints limited to τ_{Active}. The (C-graph) loop computes at each time, the new formula and labels that are considered active after the computation of Count-RT$(D)(a)$ for the previous restriction.

The next lemma shows that Algorithm Count-RT terminates.

Lemma A.12. *Let $D = (\tau, \Sigma, W, r)$ be an unordered-DTD. Then Algorithm Count-RT applied to D terminates.*

Proof. First remark that the sets τ_{Active}, N_{Active}, E_{Active} can only increase and they have finite bounds. Therefore they reach a stable value. Thus, Algorithm Count-RT terminates iff each loop C-value terminates. We demonstrate now that each loop C-value terminates. First remark, the formula $\sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b \in \lambda(\tau)} Count(b, \varphi_i)$ terminates. The formula $Count(a, \varphi)$ could raise a problem for Count-RT$(D)(a) = \infty$ if the term associated to a in φ is $|a| > max(\tau)$. However, remark that in this case, $Count(a, \varphi) = \infty$ iff Count-RT$(D)(a) = \infty$. So $Count(a, \varphi)$ can be computed in finite time. To demonstrate that the C-value loop terminates always, we prove the following property.

(†) Let N_{Active}, τ_{Active} and E_{Active} be a set of label, a restriction of τ, the set of edges derived from F such that (N_{Active}, E_{Active}) is a DAG. Then the C-value loop using the values N_{Active}, τ_{Active} and E_{Active} always terminates.

We denote by $d(a)$, the depth of a, the maximum of the length of paths from a leaf of (N_{Active}, E_{Active}) to a. We demonstrate by induction on k, that each Count-RT(a) with $d(a) \leqslant k$ is stabilized after $k + 1$ steps of the loop C-value. For the initialization, $d(a) = 0$ implies that a is a leaf. So, the labels used in the formula in $\tau_{Active}(a)$ are already computed. Then after the first step, the value of Count-RT$(D)(a)$ cannot be changed. For the propagation, we suppose that for each label b of depth less than k, then the value Count-RT$(D)(b)$ does not change after $k + 1$ iterations of the C-value loop. Let a be a label such that $d(a) = k + 1$. Then, the predecessors of a have a depth at most equal to k. Then after $k + 1$ steps, for each predecessor b of a, Count-RT$(D)(a)$ is stable by hypothesis of induction. It implies that Count-RT$(D)(a)$ is stable after $k + 2$ steps. That concludes the proof of the property.

We conclude the proof of Lemma A.12. After each step of the loop, a number Count-RT$(D)(a)$ can increase strictly only a finite number of times. First, if each label a appearing in an active strong connected component, then Count-RT$(a) = \infty$. It implies that Count-RT$(D)(a)$ does not change between two following steps of the loop C-value. Each other label a does not belong to an active strong connected component. By removing the strong components to (N_{Active}, E_{Active}), is a DAG. Property (†) shows the termination of the loop C-value for a DAG. □

Algorithm A.3 Algorithm Count-RT

Data: An unordered-DTD (τ, Σ, W, r)
Result: $\{\text{Count} - \text{RT}(D)(a) \mid a \in \Sigma\}$
begin
  foreach $a \in \Sigma$ **do**
    Count − RT$(D)(a) = 0$ $\tau_{Active}(a) = \emptyset$
  foreach $w \in W$ **do**
    Count − RT$(D)(w) = \infty$
  Count − RT$(D)(dom) = \infty$
  changedS = true
  (C-graph) loop
  while *changed* **do**
    $OldN_{Active} = N_{Active}$
    $OldE_{Active} = E_{Active}$
    foreach $a \in \Sigma$ **do**
      $Old\tau_{Active}(a) = \tau_{Active}(a)$
      foreach $\varphi \in \tau(a)$ **do**
        if $\varphi \wedge_a |a| \leqslant$ Count-RT$(D)(s)$ *is satisfiable* **then**
          $\tau_{Active}(a) = \tau_{Active}(a) \cup \varphi$
      if *Count-RT(D)(a) >0* **then**
      $N_{Active} = N_{Active} \cup \{a\}$
    foreach $(a, b) \in \Sigma^2$ **do**
      if $\exists \varphi \in \tau_{\text{Active}}(b), |a|\text{op}k \in \varphi, (|a|\text{op}k) \neq (|a| = 0)$ **then**
        $E_{Active} = E_{Active} \cup (a, b)$
    changedS = $E_{Active} \neq OldE_{Active} || N_{Active} \neq OldN_{Active} || \tau_{Active} \neq Old\tau_{Active}$
    (C-value) loop
    changedV = true
    while *changedV* **do**
      changedV = false
      foreach $a \in N_{\text{Active}}$ **do**
        OldCount-RT$(D)(a)$ = Count-RT$(D)(a)$
        if $\exists$ *a cycle in* $(N_{\text{Active}}, E_{\text{Active}})$ **then**
          Count-RT$(D)(a) = \infty$
        else
          Count-RT$(D)(a) = \sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b \in \lambda(\tau)} Count(b, \varphi_i)$
        if *OldCount-RT(D)(a)* $\neq$ *Count-RT(D)(a)* **then**
          changedV = true
end

To conclude the dicsussion about Algorithm Count-RT, we prove that it computes the guessed values $[a]_\tau$. This is proven in Lemma A.14. To prove it, we use the following Lemma.

Lemma A.13. *Let $D = (\tau, \Sigma, W, r)$ be a disjunctive DTD. At the end of each step of C-graph, for each $a \in N_{\text{Active}}$, $[a]_{\tau_{\text{Active}}} = \textit{Count-RT}(D)(a)$.*

Proof. The proof is by induction.

At the first step, the only active conjunctive formula are of the form $\wedge_{a\in\Sigma}|a| = 0 \wedge |dom|$ op $k \bigwedge_{w\in W} |w|$ op k'. In this case, it is easy to demonstrate that $\textit{Count-RT}(D)(a) = [a]_{\tau_{Active}}$.

In the propagation case, we distinguish two cases:

1. For each label a in the cycle, the number of reduced trees t rooted by a and satisfying the constraints of τ is infinite.

2. For each a that does not belong to a cycle of (N_{Active}, E_{Active})

$$[a]_{\tau_{Active}} = \sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b' \in \lambda(\tau')} \overline{Count}_{\tau_{Active}}(b', \varphi_i).$$

   where

where $\overline{Count}_{\tau_{Active}}(a, \varphi)$ is defined as follows:

- For $(|a| = k) \in \varphi$, $\overline{Count}_{\tau_{Active}}(a, \varphi) = \binom{[a]_{\tau_{Active}}}{k}$
- For $(|a| > max(\tau)) \in \varphi$, $\overline{Count}_{\tau_{Active}}(a, \varphi) = \sum_{i=max(\tau)+1}^{[a]_{\tau_{Active}}} \binom{[a]_{\tau_{Active}}}{i}$

We demonstrate Case (1) by proving the following property by induction on k:

(‡) The graph (N_{Active}, E_{Active}) has a cycle. Then for each label a in a cycle, for each k, there is a reduced tree satisfying the children constraints of τ_{Active} of root labeled a and of depth greater than k.

By definition of N_{Active}, there exists a tree of t satisfying the children constraints of τ_{Active} and having a root labeled a. This initializes the induction. Let suppose that ‡ holds for some k. Let b be a label such that $(b, a) \in E_{Active}$. Let φ be a formula of $\tau_{Active}(a)$ such that the term $|b|$ in φ is different from $|b| = 0$. By induction hypothesis, there exists a forest F of reduced trees satisfying the children constraints of τ_{Active} such that the multiset of labels of the root of the forest satisfies φ. Remark there is at least a tree t' of F with its root labeled b. Because of the recurrence hypothesis, there exists a tree satisfying the children constraints of τ, having a root labeled b and having a depth greater than k. If t belongs to F then the tree $a[F]$ satisfies children constraints of τ and has a depth greater than $k + 1$. Otherwise, we replace t' by t in the forest F given the forest F'. This forest have the same number of trees and then $a[F']$ satisfies the children constraints and has a depth greater than $k + 1$. That ends the proof of Property ‡.

Case (1) is a direct corollary of Property ‡.

We prove Case (2) by induction. Without loss of generality, we assume that the graph (N_{Active}, E_{Active}) is DAG (the cycles are removed). The initialization deals with with two cases only labels a such that the only conjunction formulas in $\tau(a)$ are on the form $\wedge_{a\in\Sigma}|a| = k \wedge |dom|$ op $k \bigwedge_{w\in W} |w|$ op k' where for each $a \in \Sigma$, Count-RT$(D)(a)$ is equal to 0 or ∞. Then it is easy to show that

$$[a]_{\tau_{Active}} = \sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b' \in \lambda(\tau)} \overline{Count}_{\tau_{Active}}(b', \varphi_i).$$

Let a be a label having some parents in (N_{Active}, E_{Active}). By induction hypothesizes, we assume that for each parent b of a that $[b]_{\tau_{Active}} = \sum_{\varphi_i \in \tau_{Active}(b)} \prod_{b' \in \lambda(\tau)} \overline{Count}_{\tau_{Active}}(b', \varphi_i)$. By using simple combinatorial reasonings, it is easy to demonstrate that $[a]_{\tau_{Active}} = \sum_{\varphi_i \in \tau_{Active}(a)} \prod_{b' \in \lambda(\tau)} \overline{Count}_{\tau_{Active}}(b', \varphi_i)$.

We conclude the propagation of the demonstration of Lemma A.13. We assume that n loops C-graph have be done. The algorithm begins a new loop C-graph. First, we want to check that for each label a in a cycle of (N_{Active}, E_{Active}), *Count-RT*$(D)(a) = \infty$. By using Property 1, it is easy to show that for each label that belongs to a graph $[a]_{\tau_{Active}} = \text{Count-RT}(D)(a)$. By a simple induction on the structure of the graph obtained from (N_{Active}, E_{Active}) by removing the cycles, it is easy to show that when the C-value loop terminates, for each a that does not belong to a cycle in (N_{Active}, E_{Active}), $[a]_{\tau_{Active}} = \text{Count-RT}(D)(a)$. □

Finally, the following lemma shows the correctness of Algorithm A.3.

Lemma A.14. *Let $D = (\tau, \Sigma, W, r)$ be an disjunctive DTD. Then for each a,* $[a] = \text{Count-RT}(a)$

Proof. For each $a \in \tau$, we denote by $\tau_{real}(a)$ the set of formulas φ of $\tau(a)$ such there is a reduced forest F verifying that $root\{F\} \vdash \varphi$ and each tree of F is rooted at a. We demonstrate by contradiction that $\tau(real)$ is equal to τ_{Active} when Algorithm Count-RT terminates.

First, remark that Property ($\sqrt{}$) holds: for a partition $\tau_1(a)$ and $\tau_2(a)$ of $\tau(a)$ then for each reduced tree satisfying the children constraints then there is a subtree satisfying only the children constraints defined by τ_1 or a subtree satisfying only the children constraints defined by τ_2.

Let assume that $\tau_{real} \neq \tau_{Active}$. We denote by τ', the set $\tau_{real} - \tau_{Active}$. Remark that (τ', τ_{Active}) is a partition of τ_{real}. By Property ($\sqrt{}$), we consider two cases:

1. There exists a reduced subtree satisfying only the children constraints of τ'.

2. There exists a reduced subtree that does not satisfy the children constraints τ_{Active} but that satifies the children constraints τ_{real} and each of its subtrees satisfies the children constraints τ_{Active}.

(1). Let us assume there exists a reduced tree t in τ'. It implies that there exists a label a and a formula φ in $\tau'(a)$ such that φ is satisfied by the multiset M, where for each $b \in \Sigma$, $M(b) = 0$. By definition of the algorithm, this formula belongs to τ_{Active} after the first C-graph loop. This is a contradiction. Then there is no reduced tree satisfying only the children constraints τ'.

(2). Let t be a tree satisfying the children constraints of $\psi(\tau)$ and that does not satisfy the children constraints of ψ_{Active}. Without losing generality, we assume that each subtree of t satisfies the children constraints τ_{Active}. Because of the correctness of Algorithm Count-RT, it means that for each a the number of subtrees of t rooted by a is less than $[a]_{\tau_{Active}}$. By definition of the algorithm, it means that the formula is satisfied by the multiset labels of the roots of the subtrees of t. Then this formula belongs to τ_{Active}. This is a contradiction.

Then τ_{real} is equal to τ_{Active} at the end of Algorithm Count-RT. Thus, by Lemma A.13, Count-RT$(D)(a)$= $[a]$ at the end of Algorithm Count-RT. □

Remark 8. Overse, that the construction blows up the size of the unordered-DTD because of the disjunctive form we use as intermediary structure. However, the following unordered-DTD shows that even without this step, the exponential blow up cannot be avoid. Let $D = (\tau, \{a_0, \cdots, a_n, b, c\}, \varnothing, a_0)$

be the following unordered-DTD:

$$\begin{array}{rcl} a_n & \longrightarrow & |a_{n-1}| \geqslant 0 \\ \cdots & & \\ a_i & \longrightarrow & |a_{i+1} \geqslant 0 \\ \cdots & & \\ a_0 & \longrightarrow & |b| = 1 \vee |c| = 1 \\ b & \longrightarrow & \\ c & \longrightarrow & \end{array}$$

By an induction, it is easy show that $[a_i]$ is equal to $\underbrace{2^{\cdots^{2^2}}}_{i}$. Thus, for any reduced DTD $D' = (\tau', \{a_0, \cdots, a_n, b, c\}, \varnothing, a_0)$ defining the same set of reduced trees than D, $\tau'(a_n)$ has to imply $|a_n| \leqslant \underbrace{2^{\cdots^{2^2}}}_{n}$. Thus, the size of D' is at least exponential in the size of D.

A.3.2. Reduced Axlog schemas

Reasoning about unordered-DTDs and reduced trees is not obvious as shown in Section 2.4. Another problem comes from the interactions between the DTD constraining the document and the specifications of trees coming from the functions. The following Axlog schema Δ shows an aspect of this problem.

$$\begin{array}{lll} d & root : r & \\ & r & \longrightarrow |w| = 1 \wedge |a| \geqslant 0 \\ & a & \longrightarrow |dom| = 1 \\ w & root : a & \\ & a & \longrightarrow \end{array}$$

A possible instance has a root r; it has a call to the function w and some a children with a data value. Observe that no tree sent from w can be inserted in the document.

Definition A.15. Let $\Delta = (d, W, \zeta)$ be an Axlog schema. Then Δ is a *reduced* Axlog schema for I iff

1. the sets of internal labels are the same in the unordered-DTDs defined in ζ; the sets of functions defined in the unordered-DTDs are equals to W.

2. for each $w \in W$, $\zeta(w)$ is a reduced DTD and $\zeta(d)$ is a reduced DTD;

3. for each $w \in W$, for each t satisfying $\zeta(w)$, t satisfies the children constraints of $\zeta(d)$.

A.3.2.1. From schema to reduced schema

The following theorem shows that for any Axlog schema, there is a reduced Axlog schema defining the behaviors..

Theorem A.16. *Let $\Delta = (d, W, \zeta)$ and I be an Axlog schema. Then, there exists an Axlog schema $Reduc(\Delta) = (d, W, \zeta')$ such that $Reduc(\Delta)$ is reduced and for each sequence ω and each I, ω is a valid sequence for Δ and I iff ω is a valid sequence for $Reduc(\Delta)$ and I.*

Proof. We give a constructive proof of the theorem. The first property implies to rewrite some constraints. Some labels considered as value (represented by *dom*) may become some labels of Σ for some unordered-DTD (τ, Σ, W, r) if this label belongs to Σ' in another unordered-DTD (τ', Σ', W, r'). Details omitted.

We handle the second property by rewriting the constraints of the unordered-DTDs describing the messages sent by the functions as follows: for each a, the constraint $\zeta(w)(a)$ is rewritten into $\zeta(w)(a) \wedge \zeta(d)(a)$. Then, we substitute the obtained unordered-DTDs by the equivalent reduced DTD. Finally, the unordered-DTD associated to the document is substituted by one of its equivalent reduced DTD. Those operations can be done using the algorithm in the proof of Theorem A.11. For each function w, $\zeta'(w)$ describes the same set of reduced trees than $\zeta(w)$, and $\zeta'(d)$ describes the same set of reduces trees than $\zeta(d)$. Then, this implies that for each sequence of updates ω and for each instance I, ω is valid for Δ and I iff ω is valid for $Reduc(\Delta)$ and I. □

A.3.3. Proofs of Theorems 2.11 and 2.12 for reduced Axlog schemas

To prove Theorems 2.11 and Theorem 2.12 for reduced Axlog schemas, we propose first an non deterministic algorithm derived from Theorem A.19 presented further. The second part of the demonstration consists in determinating the algorithm by bounding the space of research of the structures used in Theorem A.19.

A.3.3.1. Algorithm

In this subsection, recall that the Axlog schema is reduced. We need the following definition :

Definition A.17. Let $\Delta = (d, W, \zeta)$ be a reduced Axlog schema and I an instance of Δ. Let $N_0, \cdots N_k$ be a sequence of multisets over $\lambda(\zeta(d))$ and n, a node of I. Then $N_0, \cdots N_k$ is an *extension* of n *compatible* with I and Δ iff

1. N_0 is equal to the multiset of the labels of the children of n in I.
2. There exists a unique $a \in \lambda(\zeta(d))$, such that $N_i(a) + 1 = N_{i+1}(a)$; for the other labels b, $N_i(b) = N_{i+1}(b)$.
3. For each N_i, there is a conjunctive formula φ_j of $\tau(\lambda(n))$ such that the multiset N_i satisfies φ_j.
4. For each $a \in \Sigma$, the call Ids labeling children of n can bring a reduced forest F of trees rooted by a such that:
   - For each reduced tree t of F, t satisfies the unordered-DTD $\zeta(w)$, where w is the service associated to the service of a call Id, child of n.
   - No tree of the forest F appears as subtree of n in I.

- The number of trees of F is equal $N_k(a) - N_0(a)$.

Intuitively, an extension is an abstraction of a possible infinite set of valid sequences of updates such that the ith update brings a tree rooted by a where a is the label such that $N_{i-1}(a) + 1 = N_i(a)$. The following lemma explains this intuition.

Lemma A.18. *Let $\Delta = (d, W, \zeta)$, I, n be a reduced Axlog schema, an instance of Δ and a node of I. Let $N_0, \cdots N_k$ and n be an extension of n compatible with I and Δ. Then, there exists a valid sequence of updates ω such that for each i, the update $(?f, t) = \omega(i)$ satisfies the following properties: the label a of the root of t satisfies that $N_{i-1}(a) + 1 = N_i(a)$ and the call Id $?f$ is a child of n.*

Proof. We explain how to build a valid sequence from the sequence $N_0, \cdots N_k$ by induction.

We suppose that there is a valid sequence ω of size i. Let a be the label such that $N_i(a) + 1 = N_{i+1}(a)$. Following Property 4 of Definition A.17, there exists a tree t belonging to F_a and not appearing in an update of the built sequence. Following Property 4 of Definition A.17, there exists a call Id $?f$, child of n, such that t satisfies $\zeta(\nu(?f))$. The i+1th update of the sequence is $add(?f, t)$. Property 3 of Definition A.17 ensures that the sequence $\omega.add(?f, t)$ is valid. □

In the following theorem, we need a refined notion of extension. We need to add some constraints on the trees brought by the call Ids. For that, we define the notion of *DTD constraints* which is a conjunction formula of terms $|\tau| \geqslant k$, where τ is a DTD. A reduced forest F of trees satisfies a DTD constraints ψ iff there exists a partition $F_1, \cdots F_m$ of F such that (i) at each term $|\tau| \geqslant k$ of ψ is associated a set F_i such that each tree of F_i satisfies τ and the cardinality of F_i is greater than k. A set F_i cannot be associated to two terms of a formula ψ.

An extension $N_0, \cdots N_k$ of n compatible with I and Δ satisfies a DTD constraints ψ iff there exists a forest F satisfying Property 4 of Definition A.17 that satisfies also ψ.

The following theorem explains which structures consider to verify the satisfiability of a query.

Theorem A.19. *Let q, $\Delta = (d, W, \tau)$ and I be a query, a reduced Axlog schema and an instance of $\zeta(d)$. The query q is satisfiable for I and Δ iff*

1. *There exists a scenario $(u, \mathcal{C}, \mathcal{P})$ of* Scen(q, I).
2. *There is a valuation ν of the variables of u.*
3. *There is a partition π of the set $\{p|\ \mathcal{P}(p) \in F\}$, $\pi = \{P_1, \ldots P_k\}$, such that for each j, for each $p, p' \in P_j$, $\mathcal{P}(p) = \mathcal{P}(p')$. We denote by* numb$(?f)(\pi)$ *the number of sets P_i such that $\mathcal{P}(p) = ?f$, $p \in P_i$, we denote this call Id* call(P_i). *For each P_i, there exists a tree of $\zeta(\nu(\text{call}(P_i)))$ satisfying the conjunction of tree-patterns $\wedge_{p \in P_i} \nu([p])$.*
4. *For each parent n of a function node appearing in $\mathcal{P}$, there is an extension $N_0(n), \ldots, N_k(n)$ of n compatible with I, Δ. Moreover, this extension satisfies $\wedge_{?f \in \text{children}(n)} |\zeta(\nu(?f))| \geqslant$* numb$(?f)(\pi)$.

So, from the previous theorem, we obtain the following nondeterministic algorithm:

1. Compute the set Scen(q, I).
2. Guess a scenario of Scen(q, I) and a partition π.

3. Check if π satisfies Property 2 using one of the algorithms proposed in [Miklau 04, Benedikt 08, David 08].

4. Guess for each parent n of a function node appearing in $\mathcal{P}$ a sequence $N_0(n), \ldots, N_k(n)$.

5. Check if $N_0(n), \cdots, N_k(n)$ is an extension of n compatible with I and Δ and that satisfies $\wedge_{?f \in children(n)} |\zeta(\nu(?f))| \geqslant numb(?f)(\pi)$. For that, we have to count for each call Id $?f$ calling the function w the number of trees satisfying $\zeta(w)$ that can be brought also by other call Ids or already in the documents. These numbers can be computed using the algorithm of the proof of Theorem A.11 over conjunction of DTDs, which is also a unordered-DTD. For instance, the number of the same reduced trees brought by two call Ids $?f$ and $?g$ can be computed by applying this algorithm to $\zeta(\nu(?f)) \cap \zeta(\nu(?g))$.

Remark 9. One can extend the techniques to consider that results of calls may themselves contain new calls. The main part difficulty is that we have to consider tree automota to represent the trees that a function can return and no unordered-DTDs. Details omitted.

The proof of Theorem A.19 is based on Lemmas A.20 and A.22. The first lemma explains that in order to prove the satisfiability for a query, and an instance of a Axlog schema, a necessary and sufficient condition is to exhibit a scenario and a valid sequence implementing this scenario. We already showed in Algorithm A.2 to find a sufficient set of scenarios $\text{Scen}(q, I)$. The difficult part here is to decide if there is a valid sequence implementing a particular scenario. The properties given by Theorem A.19 are necessary and sufficient to check if a scenario is implemented by a valid sequence of updates. The second lemma explains that to find a valid sequence for a scenario, it is sufficient to consider only some particular subsequences. For this last lemma, we introduce the notion of sequences closed by siblings call Ids (Definition A.21).

The following lemma explains that to check that a query q is satisfiable for an instance I of a reduced Axlog schema, it is necessary and sufficient to exhibit a valid sequence of updates implementing a scenario of $\text{Scen}(q, I)$.

Lemma A.20. *Let q, $\Delta = (d, W, \tau)$ and I be a query, a reduced Axlog schema and an instance of $\zeta(d)$. The query is satisfiable for I and Δ iff there exists a scenario $(u, \mathcal{C}, \mathcal{P})$ of $\text{Scen}(q, I)$ and a valid sequence ω such that ω implements $(u, \mathcal{C}, \mathcal{P})$.*

Proof. $\Leftarrow$ Let ω, $(u, \mathcal{C}, \mathcal{P})$ be a valid sequence for I and Δ, and a scenario of $\text{Scen}(q, I)$. The sequence ω implements the scenario $(u, \mathcal{C}, \mathcal{P})$ then, by Lemma A.4, there is a tuple u' such that the document $\omega(I)$ satisfies $q(u)$. Moreover, the sequence is valid so the query is satisfiable.

$\Rightarrow$ Now suppose q is satisfiable for I and Δ. Then, there is a tuple u and a valid sequence ω such that $\omega(I) \models q(u)$. So by Lemma A.4, there exists a scenario $(u', \mathcal{C}, \mathcal{P})$ such that ω implements $(u', \mathcal{C}, \mathcal{P})$.

□

We introduce a property for the subsequences called *closed by sibling Ids* such that the validity of a sequence is equivalent to the validity of its closed by sibling call Ids subsequences.

Definition A.21. Two call Ids $?f$ and $?g$ are *siblings* iff the function nodes n and n' associated to $?f$ and $?g$ ($\lambda(n) = ?f, \lambda(n') = ?g$) are siblings. Let ω, I be a sequence of updates and an instance. A subsequence ω' is *closed by sibling call Ids* for I iff for each update $add(?f, K)$ in ω', for each update $add(?g, K')$, where $?f$ and $?g$ are sibling call Ids then $add(?g, K')$ in ω appears in ω'.

The following lemma shows that the validity of a sequence depends only of the validity of the subsequences *closed by siblings call Ids* in the document.

Lemma A.22. *Let ω be a sequence of updates. Let Δ and I be an Axlog schema and an instance of this Axlog schema. Then ω is a valid sequence of updates for the instance I constrained by the Axlog schema Δ iff for each subsequence closed by sibling call Ids ω', ω' is valid.*

Proof. $\Rightarrow$ The proof is by induction. The lemma is true for an empty sequence. Let ω and ω', I and $\Delta = (d, W, \zeta)$ be two sequences of updates, an active document and an Axlog schema. The document is an instance of $\zeta(d)$. The sequence ω is valid for I and ζ and the sequence ω' is closed by siblings call Ids subsequence of ω. We denote $last_\omega$ the last update of ω, and $\omega - \omega'$ the sequence consisting of the updates belonging to ω and not belonging to ω'. We distinguish two cases :

1. $last_\omega \neq last_{\omega'}$. Then ω' is subsequence of $\omega - last_\omega$ that is valid. By the hypothesis of induction (ω' is closed by sibling call Ids for $\omega - last_\omega$), ω' is valid
2. $last_\omega = last_{\omega'}$. Then, we consider first $\omega' - last_{\omega'}$ which is closed by sibling call Ids for $\omega - last_\omega$. So, the sequence $\omega' - last_{\omega'}$ is valid (because of the induction hypothesis). We consider $add(?f, K) = last_\omega$. Let n be the parent of the function node labeled $?f$. The set of siblings of $?f$ respects the constraints associated to their parent before and after of the addition of $root\{K\}$ (this is implied by the fact that ω is a valid sequence). So, the update ω' is valid.

$\Leftarrow$ Let ω' be a subsequence closed by sibling call Ids of ω. Observe that the subsequence $\omega - \omega'$ is also closed by sibling Ids. Then, the subsequence $\omega - \omega'$ is valid. After applying the subsequence ω', the parent nodes of call Id appearing in $\omega - \omega'$ have the same children than before applying the subsequence ω'. It is easy to check that $\omega' \cup (\omega - \omega')$ is valid.

□

Now, we demonstrate Theorem A.19.

Proof. Let $q, \Delta = (d, W, \zeta), I$ be a tree-pattern query, an Axlog schema and an active document. The document is an instance of $\zeta(d)$.

The proof uses Definition A.21 and Lemmas A.20 and A.22.

$\Rightarrow$ There exists a scenario $s = (u, \mathcal{C}, \mathcal{P})$ and a valid sequence ω valid that implements s (with Lemma A.20). Then, for each $p, \mathcal{P}(p) = ?f$, there is an update $add(?f, K)$ of ω where $K \models [p]$. Without loss of generalities, we can assume that there does not exists any update $add(?f, K)$ appearing twice or more in ω.

Let $add(?f, K)$ be an update in ω. We define by $Q(?f, K)$ the set of the nodes p of q, such that $[p]$ is satisfied by K and $\mathcal{P}(p) \in \mathcal{F}$. From the subsets $Q(?f, K)$, it is possible to find the desired partition: if there exists K and K' such that $add(?f, K)$ and $add(?f, K')$ and $Q(?f, K) \cap Q(?f, K') \neq \emptyset$, a node belonging to the two sets is removed from one. This rule is applied until there is no K and K' such that there is $?f$, $add(?f, K)$ and $add(?f, K')$, and $Q(?f, K) \cap Q(?f, K') \neq \emptyset$. These yields a partition of the nodes p such that $\mathcal{P}(p) \in F$. This partition is denoted $\{P_1, \ldots, P_k\}$. This partition satisfies Properties 3 of Theorem A.19, following the definition of $Q(?f, K)$. For each set P_i, there is an update $add(?f, K)$ such that for each $p \in P_i$, $K \models [p]$. Moreover K satisfies $\zeta(\nu(?f))$ because ω is valid.

Let n be a parent node of a call Id appearing in $\mathcal{P}$. Let ω_n be the subsequence of ω by keeping only the updates corresponding to children of n. From this subsequence, we build the sequence N_i such that $N_{i+1} = N_i \cup root\{K\}$, where $\omega_n(i) = add(?f, K)$. The multiset N_0 equals $children(n)$. This sequence is an extension of n compatible with I and Δ because ω is valid. Moreover, by construction of P_i, this extension satisfies the DTD constraints as defined in Theorem A.19.

$\Leftarrow$ Let $\pi = P_1, \ldots, P_k$ and $\{N_i\}$ be a partition satisfying Property 2 of Theorem A.19 and for each n parent of a function node appearing in $\mathcal{P}$, the sequence $N_0(n), \cdots, N_k(n)$ is an extension of n for I and Δ satisfying the DTD constraints $\wedge_{?f \in children(n)} |\zeta(\nu(?f))| \geqslant numb(?f)(\pi)$. First, we built a subsequence of updates for each n parent of a function node appearing in $\mathcal{P}$, denoted ω_n. It is built from the sequence $N_1(n), \ldots, N_k(n)$. We denote by a_i the label such that $N_i(n)(a_i) = N_{i-1}(n)(a_i) + 1$. We denote by $Sat(a, n, j)$, the set of P_i such that $call(P_i)$ is a child of n, the query associated to P_i is not satisfied by j first updates of ω_n and the label of the roots of trees brought by $call(P_i)$ is a. For each $N_j(n)$, we do the following:

- If $Sat(a_j, n, j-1) \neq \emptyset$, then a set P_i is chosen in $Sat(a_j, n, j-1)$. The update $add(?f, K)$ is added to the sequence where K satisfies $\bigwedge_{p \in P_i}[p]$ under the Axlog schema $\zeta((\nu(?f)))$ and $?f = call(P_i)$. The tree K is reduced. We can assume without loss of generality that this tree is not isomorphic to one of the subtrees of n in $\omega(0).\cdots.\omega(j-1)(I)$. Let assume that it is not the case. Then, this implies that
  - There is a scenario $(u', \mathcal{C}', \mathcal{P}')$ in $\text{Scen}(q, I)$ such that for each $p \in P_i$, $\mathcal{P}'(p) = \star$. In this case, we consider no more $(u, \mathcal{C}, \mathcal{P})$ but $(u', \mathcal{C}', \mathcal{P})$.
  - Or, there is another partition such that P_i can be brought together with the other set P_j for which the tree was added. In this case, we consider this new partition.

  Remark that in the first case, the new scenario has more nodes p of q such that $\mathcal{P}(p) = \star$. So, this argument can be only applied a finite number of times. In the second case, the new partition has less subsets than before. So, this argument can be only applied a finite number of times.

  Moreover, $\$Sat(a_j, n, j) = Sat(n, j-1) - \{P_i\}$ and for each $a \neq a_j$, $Sat(a, n, j-1) = Sat(a, n, j)$.
- If $\$Sat(a_j, n, j-1) = \emptyset$, then the update $add(?f, K)$ is added to the sequence. The tree K satisfies that the label of its root is equal a_j. The tree K belongs to $L(\zeta(\nu(?f)))$. The tree K is a fresh reduced tree. This last property is implied by the fact the DTD is a reduced DTD. Indeed, the set $N_k(n)$ satisfies φ of $\zeta(d)$, so there exists a reduced set of trees.

Now, we prove by induction that the prefixes of each $\omega(n)$ are valid. It is true for the empty subsequence. Let ω' and ω'', two prefixes of $\omega(n)$, $\omega'' = \omega' - last_{\omega'}$. We suppose ω'' valid. There is an associated sequence $N_0(n), \ldots, N_j(n)$ to ω'' by construction. Remark that $\omega'(I)$ belongs to $L(\zeta(d))$. Indeed, first, for each node $n' \neq n$ of I, the multiset of the labels of its children satisfies the constraints associated to $\lambda(n')$. Then, the multiset $N_j(n)$ satisfies the constraint associated to $\lambda(n)$ by $\zeta(d)$. Finally, as ω'' is valid then ω' is valid.

Let ω be the union of the sequences $\omega(n)$. Each $\omega(n)$ is a subsequence of ω closed by sibling call Ids. So, ω is valid. Furthermore, the sequence ω implements a scenario of $\text{Scen}(q, I)$.

□

A.3.3.2. Complexity

Theorem A.19 gives a method to check the satisfiability of a query q for an instance I of a schema Δ. We next analyze the data complexity of this method. The combined complexity is studied further.

Data complexity Theorem 2.11 states that the satisfiability of query for an instance of an Axlog schema is in PTIME in the size of the instance. We prove this theorem in the context of a particular kind of Axlog schema, the reduced Axlog schema. This proof is based on Theorem A.19. We explore the scenario of Scen(q, I) and check the properties given by Theorem A.19. Property 1 of Theorem A.19 does not depend on the size of the instance. The next lemma, namely Lemma A.23, is used to check Property (1) to (4) of Theorem A.19 by providing bounds on the size of the sequences $\{N_i(n)\}$ to check.

Lemma A.23. *Let q, $\Delta = (d, W, \tau)$ and I be a query, a reduced Axlog schema and an instance of $\zeta(d)$. The query q is satisfiable for I and Δ iff there exists a valid sequence of updates ω of length bounded by $|q| * (|q| + (|\Delta| + 1)^2)$ and there exists an instantiation of the variables h, $\omega(I) \models h(q)$.*

Proof. We denote by $\lambda\{children(n, \omega'(I))\}$ the multiset of the labels* of the children of n in $\omega'(I)$.

$\Rightarrow$ There exists valid sequence of updates ω and a scenario $s = (u, \mathcal{C}, \mathcal{P})$ of Scen(q, I), such that ω implements s. More precisely, there exists an instantiation h of the variables of q such that ω implements $h(s)$. In the rest of the proof, we consider the scenario $h(s) = (h(u), h(\mathcal{C}), \mathcal{P})$.

First, we demonstrate that the only interesting updates are those brought by call Ids siblings of call Ids appearing in $\mathcal{P}$. We denote the sequence

$$util(\omega, h(s)) = \{add(?f, K) | add(?f, K) \in \omega \wedge \exists p, \mathcal{P}(p) \text{is sibling of } f?\}.$$

No updates related to a function appearing in $\mathcal{P}$ is removed so, $util(\omega, h(s))$ implements $h(s)$. Moreover, $util(\omega, h(s))$ is closed by siblings call Ids for ω and I, so $util(\omega, h(s))$ is valid. So $util(\omega, h(s))(I)$ satisfies $h(q)$.

Then, we demonstrate that for each n, parent of a call Id appearing in $\mathcal{P}$, a valid subsequence called $rev(n)$ can be extracted from $util(\omega, h(s))$ such that $\cup_n rev(n)$ implements $h(s)$. Moreover, each sequence $rev(n)$ has a length bounded by $|q| + (|\Delta| + 1)^2$. Let n and ω_u^n be a parent of a call Id appearing in $\mathcal{P}$ and the subsequence of $util(\omega, h(s))$ composed of the updates brought by the call Ids children of n. Suppose that ω_u^n has a length greater than $|q| + (|\Delta| + 1)^2$ (in the other case, $rev(n)$ is equal to ω_u^n). Let φ be the conjunctive formula of $\tau(\lambda(n))$ such that $\lambda\{children(n, \omega_u^n(I))\}$ satisfies φ. Remark that, there exists at least one label b in $\Sigma \cup \{dom\} \cup \mathcal{W}$ such the term $|b| > max(\tau)$. So, let $\bar{\omega}_{n,h(s)}$ be the minimal valid subsequence of ω_u^n such that $\lambda\{children(n, \bar{\omega}_{n,h(s)}(I))\}$ satisfies φ. Remark that the length of $\bar{\omega}_{n,h(s)}$ is almost $(|\Delta| + 1)^2$. Moreover, for each p in q such that $\mathcal{P}(p)$ is a child of n, we choose an update $add(\mathcal{P}(p), K)$ of ω_u^n such that $K \models [p]_{h(s)}$. We denote $u(p, \omega)$ this update. Let consider that the sequence $\bar{\omega}_{n,h(s)}.\cup_{\mathcal{P}(p) \text{ child of } n} u(p, \omega)$. It is valid, in particular $\lambda\{children(n, \bar{\omega}_{n,h(s)}.\cup_{\mathcal{P}(p) \text{ child of } n)} u(p, \omega)(I))\}$ satisfies φ. This is proved by induction on ω'. We present here only the induction part. We can remark that if $u(p, \omega)$ appears in ω' then $\omega'(I)$ is equal to $\omega'.u(p, \omega)(I)$. The last tree is not reduced and its reduced associated tree is

*We assume that the labels belonging to $\mathcal{L} - \Sigma$ are replaced by the label *dom*. The call Ids are replaced by the name of the functions that they call.

equal to the reduced tree of $\omega'(I)$. If $u(p,\omega) = add(\mathcal{P}(p), K)$ does not appear in ω'. So, the term $|root\{K\}| > max(\tau)$ appears in φ and $\lambda\{children(n, \omega'.u(p,\omega)(I))\}$ satisfies φ. So, the sequence $\omega'.u(p,\omega)$ is valid. In this case, we use $\bar{\omega}_{n,h(s)}.\cup\mathcal{P}(p)$ child of $n)u(p,\omega)$ for the sequence $rev(n)$.

Finally, each sequence $rev(n)$ is closed by sibling call Ids, so $\bigcup_n rev(n)$ is valid. For each p, $\mathcal{P}(p) \in \mathcal{F}$, there exists an update $add(\mathcal{P}(p)), K)$ such that $K \models [p]_{h(s)}$, so $\bigcup_n rev(n)$ implements $h(s)$. Moreover, the number of parents of functions appearing in $\mathcal{P}$ is bounded by $|q|$. So, the sequence $rev(n)$ has a length bounded by $|q| * (|\Delta + 1|^2 + |q|)$.

$\Leftarrow$ This direction is obvious almost by the definition.

□

The proof of Theorem 2.11 for reduced Axlog schema.

Proof. It is enough :

- To enumerate all the scenarios, bounded by a polynomial function of the size of I.
- For a scenario $(u, \mathcal{C}, \mathcal{P})$, to enumerate all the partitions $\{p|\ \mathcal{P}(p) \in F\}$, the number is independent of the size of I.
- To find one partition that satisfies Property 3 of Theorem A.19. This does not depend of the size of I.
- To enumerate for each parent of a call Id appearing in $\mathcal{P}$, the sequences $N_0, \cdots N_j(n)$. The number of parents is bounded by q. Each sequence has length less than $(\Delta + 1)^2 + q$. So the maximal number of sequences does not depend of the size of I.
- To find a sequence satisfying Property 4 of Theorem A.19 can done in a polynomial time of the size of I.

That concludes the proof of Theorem 2.11 for reduced Axlog schema. □

Combined complexity We demonstrate now that the document satisfiability for a query q, an instance I of a reduced Axlog schema is NP-COMPLETE.

Proof. NP-HARDNESS For no-join Boolean queries, NP-HARDNESS is proved by reduction of the satisfiability problem of a no-join Boolean query over a fixed DTD, that is known to be NP-COMPLETE. See Theorem 4.5 of [Benedikt 08].

NP Finally, we see that we can exhibit a scenario $s = (u, \mathcal{C}, \mathcal{P})$, a valuation of the variables θ of the scenario, a partition of $\{p|\ \mathcal{P}(p) \in F\}$, a set $\{K_i\}$ of trees bounded by a polynomial function of q and a set sequence of multisets $\{N_i(n)\}$, where n is a parent of call Id appearing in $\mathcal{P}$. The length of each sequence $N_0(n), \cdots, N_k(n)$ is bounded by a polynomial function in the size of the Axlog schema, and the query. We have to check Property 4 of Theorem A.19 for each $N_0(n), \cdots N_k(n)$. The fact that a sequence $N_0(n), \cdots, N_k(n)$ is an extension of n compatible with I and Δ can be tested in polynomial time in the size of the input. Algorithm Count-RT is used to ensure the satisfaction of the DTD constraints implied by the scenario and the partition as defined in Property 4 of Theorem A.19. Finally, the set $\{K_i\}$ exhibits trees

certifying Property 3. Those trees are bounded by a polynomial size of q using Theorem 4.5 of [Benedikt 08]. As shown in Theorem 4.5 of [Benedikt 08], those trees do not belong to the language defined by the DTD. However, they can be extended by adding subtrees such that the trees belong to the language. Because the Axlog schema is reduced, we can directly use the same trees as Theorem 4.5 of [Benedikt 08]. So, we can check the satisfiability of q for I and Δ in polynomial time. This shows the membership in NP of the satisfiability of q, I and Δ for reduced Axlog schemas.

□

A.4. Nonmonotonicity

A.4.1. Noninflationary documents: Theorem 2.13

For noninflationary documents Theorems 2.12 and 2.11 do not hold. We have an analogue theorem; namely Theorem 2.13.

A.4.1.1. Upper bound:

We prove that the satisfiability problem for a query q and an instance I constrained by a schema Δ in presence of additions and deletions is Σ_2^P in the size of I and q.

Proof. First, we define a set of tuples $(\omega, I_0, \cdots, I_k, d, \nu, val_1, val_2)$ that we denote L as follows:

- ω is a sequence of updates of which the size is bounded by $|q| * (|q| + (1 + |\Delta|)^2 + |I|))$, I_i is an instance,
- d is function that for update $w(i) = delete(?f,q)$ gives the nodes of the initial instance and the trees adding updates that are deleted by it.
- ν is a instantiation of the variables of q.
- $val_j(i)(n)$ defines an instantiation oj the variables of q_i such that $\omega(i) = delete(?f, q_i)$ associated to a node n.

The tuple $(\omega, I_0 \cdots I_k, \nu, val_1, val_2)$ belongs to L iff

1. $I_0 = I$
2. Each I_i can be extended to an instance satisfying Δ.
3. I_k satisfies $\nu(q)$.
4. for all i, if $\omega(i) = add(?f, K)$ then $I_{i+1} = \omega(i)(I_i)$.
5. if $\omega(i) = delete(?g, q_i)$ then
   - for each $n \in d(\omega(i)) \cap I$, then $n \in I_i$ and $n \notin I_{i+1}$, n is a sibling of $?g$ and $\lfloor n \rfloor \models val_1(i)(n)(q_i)$
   - for each $n \in I_i \cap I$ and $n \notin d(\omega(i))$, then $n \in I_{i+1}$. Moreover, if n is a sibling of $?g$, $\lfloor n \rfloor \not\models val_2(i)(n)(q_i)$ and $\lfloor n \rfloor \not\models val_1(i)(q_i)$;
   - for each tree $K \in d(\omega(i))$, then K is in I_i and not in I_{i+1}

- for each tree K brought by an update, such that K is in I_i and not in $d(\omega(i))$ then K is in I_{i+1}.

The size of a tuple is polynomial in the size of the query and the initial instance. Remark that the queries in a deletion operation can be bound by a polynomial size of the instance I.

One can verify that a tuple is in L in PTIME in the size of the query and the initial instance. The main part is to check that there are enough different reduced trees brought by updates that have to be or not be deleted after. For that, we count the trees of satisfying queries under Δ using an adaptation of Algorithm Count-RT.

We denote by L' the language of the set of tuples $(\omega, I_0 \dots I_k, \nu, val_1)$ such that

$$\forall val_2, (\omega, I_0 \dots I_k, \nu, val_1, val_2) \in L$$

.

One can show that a tuple belongs to L' iff there is a valid sequence of updates ω such that $\omega(I) \models q$:

Proof. $\Leftarrow$ Let ω be a valid sequence of updates such that $\omega(I) \models q$. So, there exists an instantiation θ of the variables of q such that $\omega(I) \models \theta(q)$. Without loss of generality, we can assume that the length of ω can be bounded by $|q| * (|q| + (1 + |\Delta|)^2 + |I|)^*$. The sequence $\{I_i\}$ is built as follows: $I_0 = I$ and $I_{i+1} = \omega(i)(I_i)$. Let $\omega(i) = delete(?f, q_i)$ and n be an update and a node in I_i and not in I_{i+1}, $\lfloor n \rfloor \models q_i$. Then there exists an instantiation $\theta_{i,n}$ of the variables of q_i such that $\lfloor n \rfloor \models \theta_{i,n}(q_i)$. Moreover, for each n in I_i and in I_{i+1}, n is a sibling of $?f$ implies that $\lfloor n \rfloor \not\models q_i$. So, val_1 is built by assigning for each node i such that $\omega(i) = delete(?f, q_i)$ and for n a sibling of $?f$ appearing in I_i and not I_{i+1} to $val_1(i)(n)$, the instantiation $\theta_{i,n}$. For the other values of i and n, an instantiation is chosen randomly. For any instanciation val_2, the tuple $(\omega, I_0, \dots, I_k, \nu, val_1, val_2) \in L$, so $(\omega, I_0, \dots, I_k, \nu, val_1) \in L'$.

$\Rightarrow$ Let $(\omega, I_0, \dots, I_k, \nu, val_1)$ be a tuple in L'. Let i be an integer such that $\omega(i) = delete(?f, q_i)$. Let n be a node in I_i and in I_{i+1} and sibling of $?f$. Then for each instantiation θ' of the variables of q_i, $\lfloor n \rfloor \not\models \theta'(q_i)$, and so $\lfloor n \rfloor \not\models q_i$. Thus, we have that for each i, $I_{i+1} = \omega(i)(I)$. Moreover, ω is a valid sequence of updates such that $\omega(I) \models \nu(q)$.

□

So q is satisfiable iff there exists $(\omega, I_0 \dots I_n, \nu, val_1) \in L'$. So the problem is in Σ_2^p. □

A.4.1.2. Lower bound:

CO-NP-HARDNESS

We demonstrate this lower bound by reducing the not 3-colorable graph problem.

Proof. Let $G = (V, E)$ be a graph without auto-loops. We reduce the problem of 3-colorability of G by the satisfiability of the query q for the instance I under the schema Δ. The definitions of the query q and the schema Δ are independant of G. Only the definition of I changes following the

*The proof of this assumption can be done by adapting the proof of Lemma A.23

definition of G. The schema Δ is the following :

$$
\begin{array}{lll}
d & root : r & \\
 & r \longrightarrow & (|a| = 3 \wedge |w_d| = 1 \wedge |w_a| = 1) \\
 & & \bigvee(|a| = 1 \wedge |b| = 1 \wedge |?g| = 1 \wedge |?f| = 1) \\
 & & \bigvee(|a| = 1 \wedge |?g| = 1 \wedge |?f| = 1) \\
 & b \longrightarrow & \\
 & a \longrightarrow & |e_1| \geqslant 0 \wedge |n| \geqslant= 0 \\
 & node \longrightarrow & |val| = 1 \\
 & e_1 \longrightarrow & |e_2| = 1 \wedge |val| = 1 \\
 & e_2 \longrightarrow & |val| = 1 \\
 & val \longrightarrow & |dom| = 1 \\
w_d & delete & \\
w_a & add \quad root : b & \\
 & b \longrightarrow &
\end{array}
$$

The function w_a can send some update $add(?f, t)$ with t is a single node labeled b and $?f$ is a call Id to w_a. The function w_d can send deletions. The query q checks whether the instance has a child labeled b of the root of the instance I. The instance I has three subtrees rooted by a. Each subtree rooted by a describes a graph as follows: the value under the node labeled *node* is the identifier of a node of the graph. A value is a label which does not belong to $\{r, a, b, node, val, e_1, e_2, \}$. Each edge (n_1, n_2) of E is represented by a subtree rooted by e_1 such that the value under the child *val* of e_1 is n_1 and the value under the child *val* of e_2 is n_2. In Figure A.2, the instance I' is an encoding of the "triangle" graph, i.e $\{x_1, x_2, x_3\}, \{(x_1, x_2), (x_2, x_3), (x_1, x_3), (x_3, x_1), (x_3, x_2), (x_2, x_1)\}$ for some x_1, x_2, x_3. Two of the subtrees, denoted t^1_G and t^2_G, rooted by a in I describe the graph G, and the third subtree, denoted t_T, rooted by a describes the triangle graph. We assume that a value does not appear in two subtrees rooted by a.

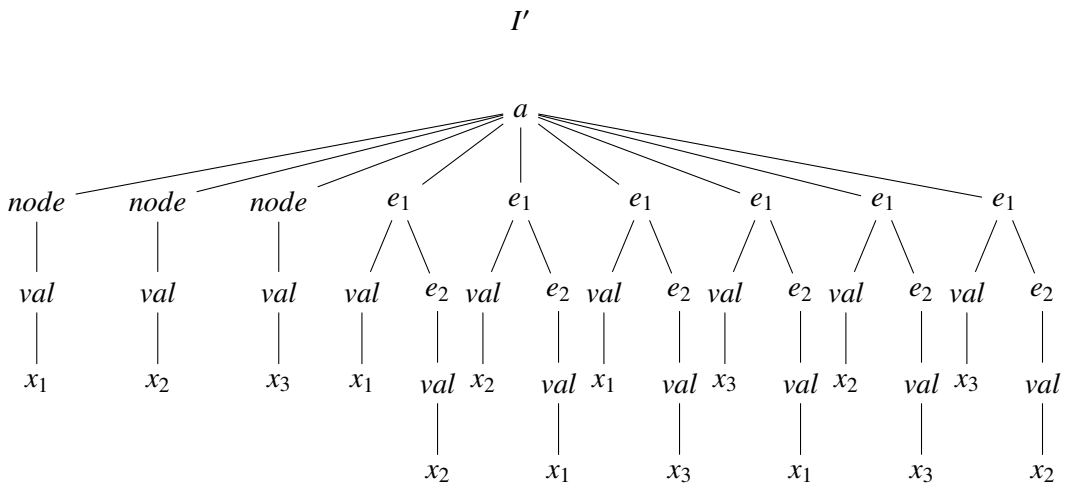


Figure A.2.: An tree coding the triangle graph

We demonstrate now that the query q is satisfiable for I and Δ iff it is possible to delete two subtrees rooted by a of I by the same deletion and only two. Indeed, if it is possible, this deletions is applied and after a node b is added by an update from w_a. If q is satisfiable then there exists a

sequence of updates such that $w(I)$ has a child b. It implies that there is only one subtree rooted by a. Then, it implies that there is an update deleting two subtrees at the same time. On the other hand, the query q is satisfiable for I and Δ iff there exists a query q' which (i) is satisfied by two subtrees of I with a root labeled by a and (ii) is not satisfied by the other tree of I rooted by a label a.

We demonstrate towards a contradiction that there does not exist a query q' satisfying Properties (i) and (ii) for I iff G is homomorphic to the triangle graph (in another hand, G is 3-colorable). Let suppose that G is homomorphic to the triangle graph, we denote by h this homomorphism. Suppose towards contradiction that a query q' that satisfies Property (i) and (ii). Then, q' does not have any value of a subtree rooted by a inside itself (a value does not appear in two subtrees of I rooted by a). So, the query uses only the labels $a, node, e_1, e_2, val$ and variables. We denote ν the valuation from q' to t^1_G. Then, we explain how to build a valuation from q' to t_T. First, we create a function ξ from t^1_G to t_T as follows (Remember that each value x appearing in t^1_G corresponds to a node x of the graph G)

- The root of t^1_G is mapped to the root of t_T
- Each subtree rooted by *node* and with value x inside is mapped to the subtree rooted by *node* and with the value $h(x)$ inside.
- Each subtree corresponding to the edge (x, y) in G is mapped to the subtree corresponding of the edge $(h(x), h(y))$ in the triangle graph.

The function ξ respects the root, child and descendant relations and for each value x of t_T, the images of nodes labeled by x have the same label $h(x)$. Then, the function $\xi \circ \nu$ is a valuation from q' to t_G. Thus, there is a contradiction.

Suppose now that there does not exists a query q' satisfying Properties (i) and (ii). Let q_G be the query obtained from t^1_G by replacing any value x of t^1_G by the variable $\$x$. This query is satisfied by t^1_G and t^2_G. Then, this query is satisfied by t_T. Let ν be a valuation from q_G to t_T. Clearly, the function associating to the variable $\$x$ appearing in q_G to the value in t_T is a homomorphism of graphs. Then G is homomorphic to the triangle graph. □

A.4.2. Non monotonic queries

In this subsection, we prove Theorems 2.15 and 2.16.

A.4.2.1. Proof of Theorem 2.15-data complexity

We proof first the data complexity of the satisfiability of time-query over an active document.

Proof. We adapt Algorithm A.2 by adding to the scenario, a generalized tuple $(u_t, \mathcal{C}_t)$. Let p and $(n, u, \mathcal{C}, u_t, C_t, \mathcal{P})$ be a node of q and a tuple of a relation $relevant_p$. The tuple u_t is a tuple of variables over the time of the nodes of $[p]$. We assume that the variable $\$t_p$ is the variable assigned to the time of the node p ($u_t(p)$). The constraints C_t are the inequalities in q dealing with the nodes appearing in the subquery p and the instantiation of the time variables of the nodes already mapped. Those constraints are expressed by inequalities constraints. Remark that by the definition of a generalized tuple $(u, \mathcal{C})$, the constraints $\mathcal{C}$ are satisfiable using the values in u for the corresponding variables.

Remark that by adding time, even if there is some scenario in $Scen(q, I)$, this does not imply the satisfiability of the query. Indeed, the time constraints and the fact that time increases with the

updates imply that the order of a sequence now is relevant. In Figure A.3, the query is unsatisfiable. The only scenario is $((), \varnothing, (\$t_{p_2}, \$t_{p_3}, \$t_{p_4}, \$t_{p_5}), (\$t_{p_2} < \$t_{p_4} \wedge \$t_{p_3} > \$t_{p_5}), (\star, ?f, \bullet, ?f, \bullet))$. Suppose that the first relevant update satisfies the subquery $[p_2]$, and the second relevant update satisfies the subquery $[p_4]$. Then the times of the updates respect the constraints $(\$t_{p_2} = \$t_{p_3} \wedge \$t_{p_4} = \$t_{p_5} \wedge \$t_{p_2} < \$t_{p_4})$. There is a contradiction with the constraints of the query. The same kind of contradiction appears by choosing as first relevant update the update satisfying $[p_4]$.

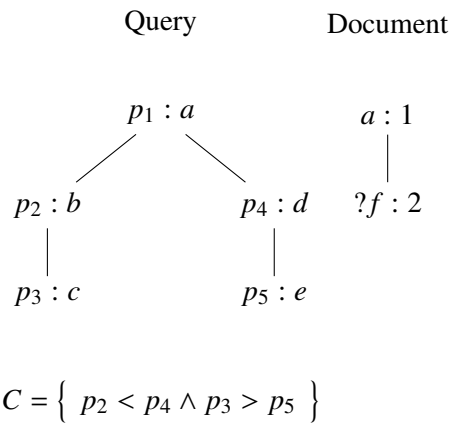


Figure A.3.: A timed query and a timed document

So, after having computed $Scen(q, I)$, each scenario $(u, \mathcal{C}, u_t, \mathcal{C}_t, \mathcal{P})$ is rewritten as following

1. Let $\pi = \{P_1, \ldots P_k\}$ be a partition of the set $\{p |\ \mathcal{P}(p) \in F\}$ such that for each P_i:
   a) For $p, p' \in P_i$, $\mathcal{P}(p) = \mathcal{P}(p') \in \mathcal{F}$, we denote by $fun(P_i)$, the call Id associated to the node of P_i by $\mathcal{P}$.
   b) the boolean combination* $\bigwedge_{p \in P_i}[p]$ is satisfiable.

   Two nodes p and p' belong to P_i means there is an update that satisfies two subqueries $[p]$ and $[p']$.

2. Let $\prec$ be a total order for π. This order is the order of the sequence of updates. An update $add(?f, t)$ such that t satisfies $\bigwedge_{p \in P_i}[p]$ and $?f = fun(P_i)$, is received before an update $add(?g, t)$ such that t satisfies $\bigwedge_{p \in P_j}[p]$ and $?f = \mathcal{P}(p), p \in P_j$ iff $P_i \prec P_j$.

3. The following constraints are added to the constraints $\mathcal{C}_t$:
   - for each $P_i \in \pi$, for each node $p, p' \in P_i$, $\$t_p = \$t_{p'}$ (the subqueries are satisfied by the same update)
   - for each $P_i, P_j \in \pi$, for each node $p \in P_i, p' \in P_j$, $P_i \prec P_j$, $\$t_p < \$t_{p'}$ (the update satisfying the queries of P_i is before the update satisfying the queries of P_j)
   - for the minimal element P_i of π for each node $p \in P_i$, P_i is the minimum for $\prec$, $\$t_p > now$ (the first updates is received after now)

*We extend the semantic of tree pattern query to the boolean combination of tree patterns queries by the classical semantic.

- for each $P_i \in \pi$, for each node $p \in P_i$, p' a descendant of p, $\$t_p = \$t_{p'}$ (the nodes of the same update have the same time)

The set of scenario that is thereby obtained is called $\overline{\text{Scen}}(q, I)$.
(†) We prove query is satisfiable iff there is scenario in $\overline{\text{Scen}}(q, I)$.

Proof of (†) Remark that the constraints C_t of one scenario of $\overline{\text{Scen}}(q, I)$ are satisfiable.

If there is a scenario $s = (u, \mathcal{C}, u_t, \mathcal{C}_t, \mathcal{P})$ of $\overline{\text{Scen}}(q, I)$, then there are a scenario s' of $\text{Scen}(q, I)$, a partition π and an order $\prec$ such that s is the rewriting of s' with π and $\prec$. Because the constraints $\mathcal{C}_t$ are satisfiable, there is a valuation θ of the variables $\$t_p$. Let ω be a sequence of updates. At each update is associated a set P_i of π. The first update is associated to the minimal P_i for $\prec$. If an update is associated to P_i, then the next update is associated to P_J, where P_j is the successor of P_i following $\prec$. The time of all the nodes of an update is $\theta(p)$, $p \in P_i$, where P_i is the set associated to the update. Then, the instance $\omega(I)$ satisfies q.

If the query is satisfiable. Then there exists ω a sequence of updates such that $\omega(I) \models q$. Let ν be valuation from the query to the instance $\omega(I)$. Let ω' be the sequence updates that brought a node mapped in ν. The time constraints for the sequences are respecting by ω' , because it is a subsequence of ω. It is easy to find a scenario in $\text{Scen}(q, I)$ and a partition π, by associating at each update, the subqueries mapped in this update. Then the scenario s is rewritten following this partition and this order of the sequence. The new constraints are satisfiable at least with the time of the nodes of ω'. So, $\overline{\text{Scen}}(q, I)$ is not empty.

The proof of † completes the proof of Theorem 2.15 □

A.4.2.2. Proof of Theorem 2.15-combined complexity

Proof. The NP-hardness of time query satisfiability is proved by reduction of NP-hardness of time query satisfaction. The time query satisfaction NP-hardness is proved by reduction of 3-Sat. Let φ be 3-Sat formula. Let $Var(\varphi)$ be the set of variables in φ. For each variable x of $Var(\varphi)$ the labels var_x, x and $\bar{x}$. The document has root labeled r which a subtree for each variable x of $Var(\varphi)$ such that the root of this subtree is var_x and four children, two labeled by x and two labeled by $\bar{x}$. The times of r, var_x, one x and one $\bar{x}$ are 0 and the time of the other x and $\bar{x}$ is 1. The associated query is exactly the document with the following constraints:

- for each variable x, $x + \bar{x} = 1$
- for each clause and the literal l_1, l_2, l_3 of the clause, $l_1 + l_2 + l_3 \geqslant 1$

It is obvious that the document satisfies the query iff the formula is satisfiable □

A.4.2.3. Proof of Theorem 2.16

We reduce the problem of implication of function dependencies (fd's in short) and inclusion dependencies (ind's in short) for relational databases to the satisfaction problem for queries with negations. (Satisfaction can then easily be reduced to document satisfiability.)

We recall classical definition of dependencies in the relational model:

fd $I \models A_1 \ldots A_m \to B$ (where A_i and B are attributes) iff for each tuple $s, t \in I$, $s(A_i) = t(A_i)$ for each i implies $s(B) = t(B)$.

ind $I \models A_1 \ldots A_m \subseteq B_1 \ldots B_m$ (where A_i, B_i are attributes), if for each tuple r in I, there exists a tuple s in I such that for each i, $r(A_i) = s(B_i)$.

Consider a relation R over $A_1 ....A_m$. We can represent it naturally as a tree with root labeled by r, with a child labeled R, that has children labeled t (one per tuple), each with m children labeled $A_1, ..., A_m$, each with a set of children. The set of labels under the node labeled A_i represent the value $t(A)$. Each value v in a tuple t is represented in our tree by a set of labels. We impose with some patterns that if a label appears in two sets then the sets are equal. So to test equality between two values of tuples, we have to test equality if the two sets have a same label. To test inequality between values of two tuples, we have to test if a label appears in one set representing the values and not in the other one. One can construct queries as follows:

1. a Boolean query γ that tests whether the document d is indeed a representation of a relation R.

2. for each dependency χ (functional or inclusion), a Boolean query $q(\chi)$ that checks whether χ is satisfied.

Let $\chi_1, ..., \chi_n$ and χ be functional or inclusion dependencies. Consider the query q consisting of a root labeled r, positive subpatterns for each χ_i, and γ, and a negative one for χ. Then one can prove that

$$\chi_1 \wedge ... \wedge \chi_n \nvdash \chi \text{ iff } \exists d(d \models q)$$

.

A.5. Relevance

We prove the np-hardness of relevance of a call Id for a Boolean query.

The proof is by reduction of 3SAT. Let $\varphi = \wedge_{i \in [1..n]} C_i$ be such a formula. The corresponding instance is constructed as follows. For each C_i, let c_i be a distinct new label. The instance also uses functions h_j^0 and h_j^1 for each x_j. The root, labeled r, has one subtree t_j for each variable x_j and one other subtree t_c. The subtree t_j has a root labeled a and two subtrees, t_j^0, t_j^1 defined as follows:

- t_j^0 has root labeled a, one children labeled c_i for each C_i where $\bar{x}_j$ occurs, and two other subtrees: one consisting of a single node labeled 0; and one subtree $a[h_j^0]$.

- t_j^1 has root labeled a, one children labeled c_i for each C_i where x_j occurs, , and two other subtrees: one consisting of a single node labeled 1; and one subtree $a[h_j^1]$.

The subtree t_c is: $a[\ a[1\ a[1]]\ a[0\ a[h]]\ ]$; where h is the function for which we question the usefulness.

The query q is constructed as follows. It has a root r with one subtree q_i for each clause C_i plus a subtree q_c, defined as follows. Query q_i has a root labeled a and a unique child

$$a[\ c_i\ a[1]\ ]$$

The other child of the root of q is

$$a[\ a[1\ a[1]\ ]\ a[\ 0\ a[1]\ ]\ ]$$

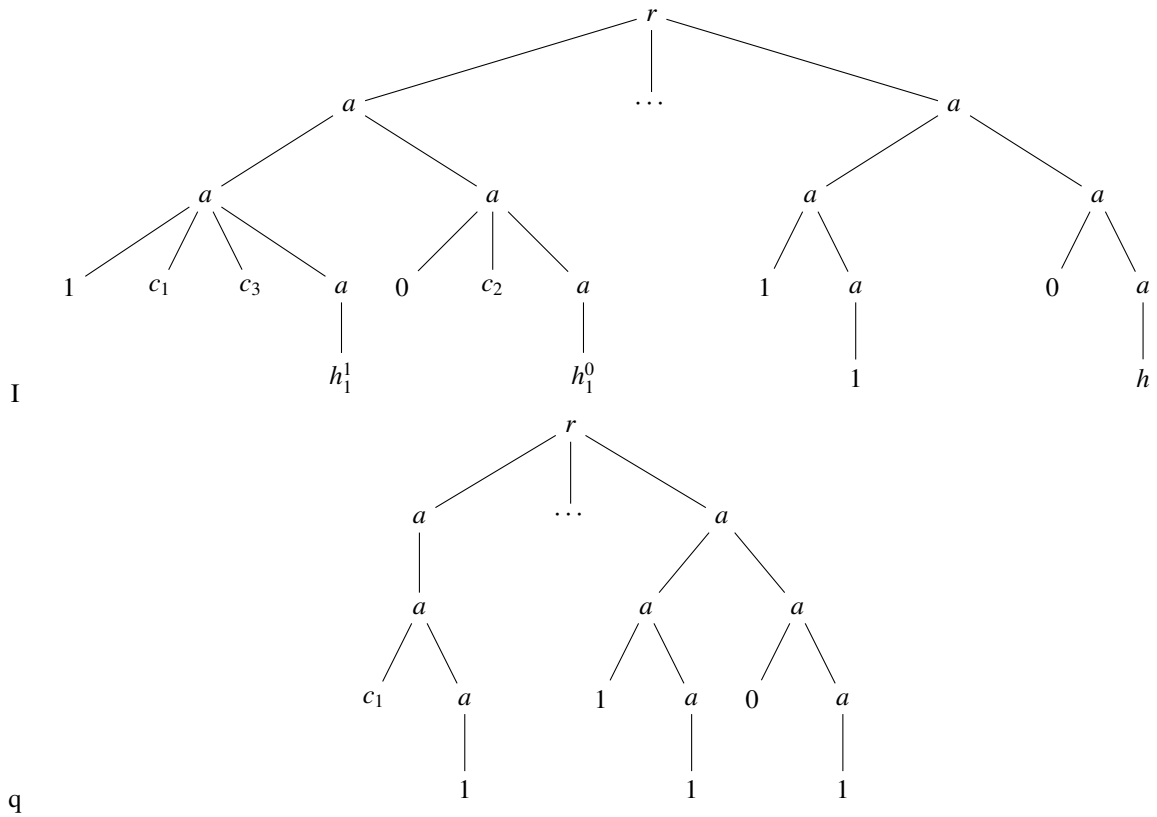


Figure A.4.: The construction for Theorem 2.18 of φ

The instance I and the query q are shown in Figure A.4 for

$$\varphi = (x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee \bar{x_2} \vee x_3)$$

One can show that φ is satisfiable iff h is useful: Suppose φ is not satisfiable. Then every instance that satisfies $\wedge q_i$ also satisfies q_c, so h is useless.

Conversely, suppose that φ is satisfiable. Let ν be a valuation that satisfies φ. Consider the update w that consists in sending a 1 to each h^1_j of $\nu(x_j) = 1$ and sending a 1 to h^0_j otherwise. Let $J = w(I)$. Then $J \not\models q$ whereas $add(h, 1)(J)$ does. Thus h is useful.

Appendix B.

Comparing Workflow Specification Languages: The Proofs

B.1. Proof of Theorem 5.7

We adapt the proof of Theorem 4.2 of [Abiteboul 09], showing that it is undecidable, given a positive pattern P without variables and a GAXML schema $S|\gamma$, whether some instance satisfying P is reachable in a valid run of $S|\gamma$.

The proof is by reduction of the implication problem for functional and inclusion dependencies (FDs and IDs), known to be undecidable (see [Chandra 85]). 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 B.1. Clearly, this structure can be enforced by the DTD.

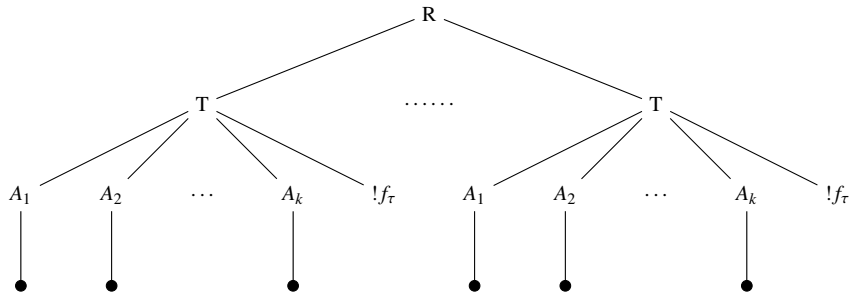


Figure B.1.: Relation adorned with some functions

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 f_τ for each $\tau \in \Gamma$. One occurrence of each f_τ is attached to each tuple of R, as in Figure B.1. The functions f_τ always return the empty answer. Static constraints require the following:

(i) there is at most one occurrence of $?f_\tau$ for each τ,

(ii) whenever $?f_\tau$ occurs, the ID τ is satisfied for the tuple to which $?f_\tau$ is attached.

The constraint (i) is expressed by conjunctions of negations of patterns as in (i) of Figure B.2 and using the fact that there are no two subtrees rooted at T representing the same tuple. This property

is due to the fact that the initial active document is reduced. The constraint (ii) is enforced by the conjunction of patterns as in (ii) of the same figure, illustrating the case when $\tau = R[A_i] \subseteq R[A_j]$.

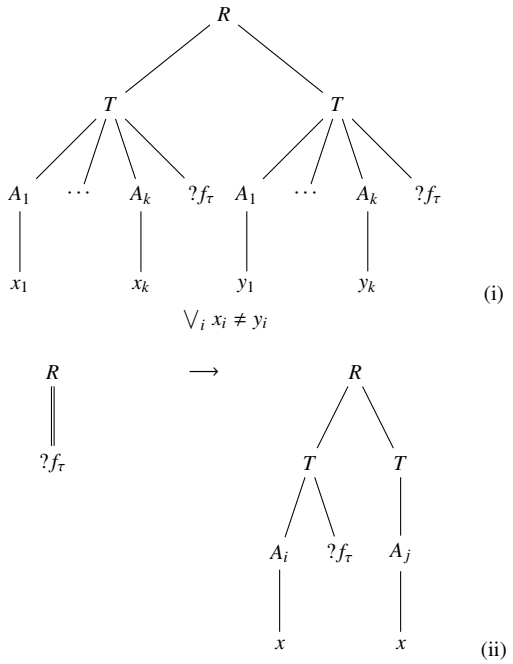


Figure B.2.: (i) Pattern whose negation forbids two activated calls and (ii) ensuring satisfaction of $[A_i] \subseteq [A_j]$

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 $!h$. Thus, the instance I_0 consisting of the root r with child $!h$ is a possible initial instance. Clearly, there exists a valid run from I_0 iff the function h can return a tree R witnessing that $\Gamma \not\models F$, which concludes the proof.

B.2. Proof of Theorem 5.9

Let $S|W$ be a workflow schema and $\rho = (I_i)$, $1 \leqslant i \leqslant n$ be 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), $j \leqslant i$, 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), $0 \leqslant j \leqslant i+1$. Therefore, the set of relations $\{R_q^n \mid q \in Q\}$ can be constructed in time polynomial in $|\rho|$. Finally, ρ satisfies A iff some relation R_q^n 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 is has no global variables. Let θ_ν^0 be a Past-LTL propositional formula from which θ_ν is obtained by interpreting the propositions by Boolean patterns 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_ν^0. From A_0 it is straightforward to construct a Tree-pattern automaton A_ν such that $S|\theta_\nu$ and $S|A$ 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.

B.3. Proof of Theorem 5.10

(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 08]. (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 08]. 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 $?f$. 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 $?f$ can return.

B.4. Proof of Theorem 5.11

Let W_1 and W_2 be workflow specifications. Observe that with the identity view, $[W_1] \sim [W_2]$ iff W_1 and W_2 have the same sets of runs. We denote the latter by $W_1 \equiv W_2$.

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 wrt homomorphism, which is not the case for guards. In the following, the view is fixed to be *id*.

B.4.1. Lemma GAXML $\not\hookrightarrow_{(id,id)}$ AAXML

Proof. We exhibit a GAXML schema $S|\gamma$ for which no pattern automata schema has the same set of runs.

We briefly describe some aspects of S. Its DTD imposes that its instances consist of a tree of root r with six children that are function calls to some internal functions $f_1, ..., f_5$ and *end*. The argument query of each f_i is f for some internal function f and its return guard is false. The argument query of f is some internal function g and its return guard is also false. 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.

At some stage, we have reached the instance I_1 where the f_i have been called as well as the function f in the workspace for f_i for each i. More precisely, the instance consists of: a tree with a root labeled r, five trees with roots labeled a_{f_i}, $i \leqslant 5$, and five trees with roots a_f. The r root has for children five activated functions calls $?f_i$, $i \leqslant 5$, and one inactive function call *end*. For each $i \leqslant 5$, the workspace of the function f_i has a root labeled a_{f_i} with for unique child, an active function call $?f$. Each workspace for a function call f has a root labeled a_f with as child, an inactive call to function g.

Now consider the following two sequences from I_1:

- $\{I_i\}$: activate 3 g's; activate *end*; return 2 g's.
- $\{J_i\}$ with $J_1 = I_1$ activate 3 g's; activate *end*; return one g; activate one g.

Observe that the first is accepted by $S|\gamma$ and not the second because the last activation of g violates its guard.

Suppose that there exists an automaton A such that $S|A \equiv S|\gamma$. Since $\{I_i\}$ is accepted by $S|\gamma$, it is accepted by $S|A$. Now, we can clearly assume that its state formulas do not have free variables since there is only a fixed set of labels in $S|\gamma$. Observe that for each i, there is a homomorphism from I_i to J_i, and conversely. They are the same except for the last one and in both last ones, some g has been activated, some g has terminated, and some have not been activated. (This is where we use the fifth g.) Thus any state formula that holds for I_i, also holds also for J_i. Since $\{I_i\}$ is accepted by $S|A$, $\{J_i\}$ is accepted by $S|A$, so by $S|\gamma$, a contradiction. Hence there is no automaton A such that $S|A \equiv S|\gamma$.

□

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

B.4.2. Lemma GAXML $\not\hookrightarrow_{(id,id)}$ TAXML

Proof. This follows by a similar observation as above: the set of runs definable by a Past-Tree-LTL formula is closed under equivalence wrt homomorphism (without data values). 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.

B.4.3. Lemma AAXML $\not\hookrightarrow_{(id,id)}$ GAXML

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. □

B.4.4. Lemma TAXML $\not\hookrightarrow_{(id,id)}$ GAXML

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

B.4.5. Lemma TAXML $\not\hookrightarrow_{(id,id)}$ AAXML

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 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 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 describing the same sequence. The state of the automaton after step (4) cannot have any parameters, since the current instance has no data value. Then the automaton 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 04].

B.4.6. Lemma AAXML $\not\hookrightarrow_{(id,id)}$ TAXML

Lemma B.1. *AAXML* $\not\hookrightarrow_{(id,id)}$ *TAXML*

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 does nothing (its return guard is false). 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 $\{(ab)^{2n}c \mid n \geqslant 0\}$. Note that L cannot be expressed by FO on words because it is not counter free [Diekert 08], so it can neither be expressed by LTL [Libkin 04]. 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 the set of instances of $S'|\theta$, $\{a, b, c\}$ or $\{a, b, c, d\}$. In the first case, ψ_Δ is $\mathbf{G}(a \vee b \vee c)$. In the second case, 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. □

B.5. Proof of Theorem 5.12

B.5.1. Simulation of GAXML by BAXML

This proof consists of two parts: 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$^{no\text{-}ret}$.

Lemma B.2. *GAXML* $\hookrightarrow_{(id,\pi)}$ *GAXML*$^{\text{no-ret}}$.

Proof. We explain how we can remove the return guards of GAXML schemas. Intuitively, we simulate the check of the return guard of a workspace of $?f$ using a function call $!check\text{-}rg_f$ in the same workspace, whose call guard checks the return guard of f. We wish to ensure the following property, while avoiding infinite branches of ε-transitions:

(+) the call to $?f$ can return only if the call to $!check\text{-}rg_f$ 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 (+) requires some auxiliary functions. Recall that, by definition, the answer cannot be returned as long as the workspace contains active function calls. In the simulation, the answer to $?f$ can be returned after the answer to $?\textit{check-rg}_f$ is returned. To inhibit other visible transitions, the answer to every other function call is forced to contain a call to an internal function $!\textit{return}$ whose answer is empty. Property (+) is mainly ensured by two relative queries added to the return query of f:

$$a_f//!\textit{check-rg}_f \longrightarrow \{\textit{error}\}$$

and

$$a_f//!\textit{return} \longrightarrow \{\textit{error}\}.$$

A constraint states that *error* may not occur in a valid instance.

We explain in more detail the sequence of actions, in particular how to ensure there are no infinite sequences of ε-transitions. First, the activation of $!\textit{check-rg}_f$ is allowed (by using the call guard) iff the return guard of f is true, there is no activated function call, and $!\textit{return}$ is not in the workspace. The function call $?\textit{check-rg}_f$ returns another function call $!\textit{rg-ok}_f$. The function call $!\textit{rg-ok}_f$ is only activated in the presence of $!\textit{return}$, and returns $!\textit{check-rg}_f$ (the presence of $!\textit{return}$ ensures that a visible transition occurs, thus preventing infinite sequences of ε-transitions). The function $!\textit{return}$ can be activated in the workspace, returning the empty answer, only if $?\textit{rg-ok}_f$ also occurs in the workspace. In summary, the call $?f$ can return only if $!\textit{rg-ok}_f$ is in the workspace and no visible transition occurred since the firing of $!\textit{check-rg}_f$. Also, consecutive calls to $!\textit{check-rg}_f$ cannot occur without the return of a visible function call in the workspace. This ensures (+) while preventing infinite sequence of ε-transitions.

Figure B.3 gives an overview of the possible sequences of function calls for the new functions.

□

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

Lemma B.3. *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 $\gamma_c(f)$ and insert a corresponding tag in the workspace, signaling satisfaction of the pattern. Specifically, for each pattern P of $\gamma_c(f)$, we add to the argument query of f a rule $P \rightarrow \{\textit{sat}_P\}$ where $\textit{sat}_P$ 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 $\textit{sat}_P$ corresponding to truth assignments satisfying $\gamma_c(f)$. This ensures that $!f$ can only be activated if $\gamma_c(f)$ 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 $!f$ is accompanied by a sibling $!\textit{lock}_f$. This is done using the DTDs (including those of answers to external function), as well as by modifying the return queries of internal functions by adding to every occurrence of $!f$ a sibling $!\textit{lock}_f$.

The function $!\textit{lock}_f$ is internal and has several roles:

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

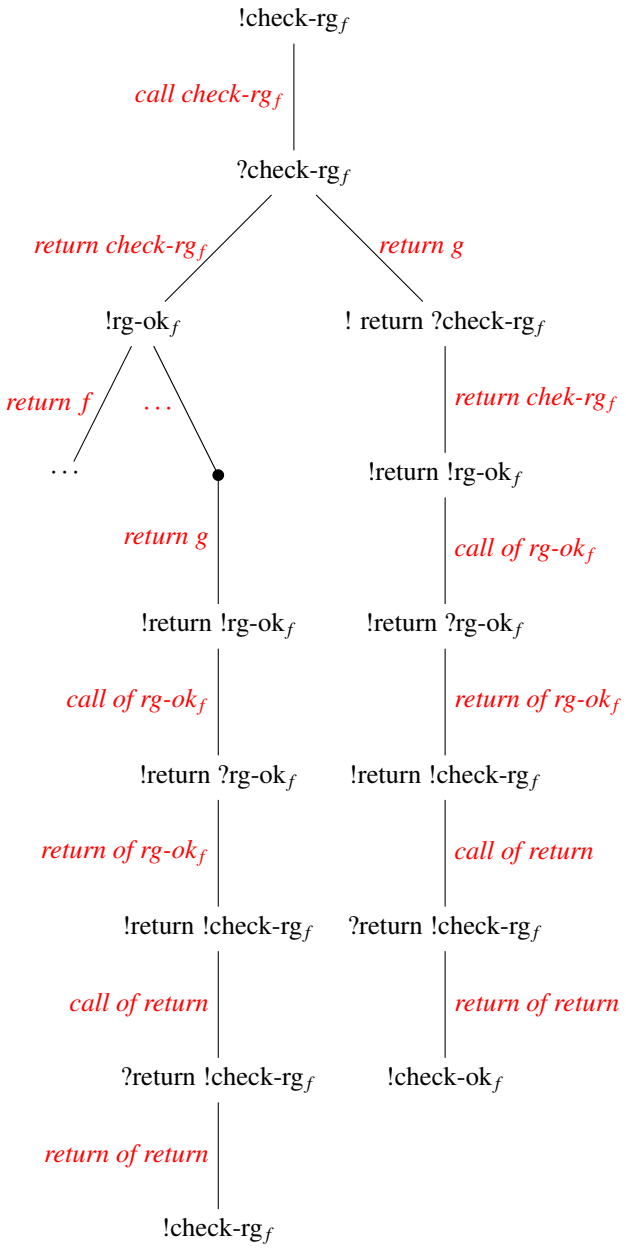


Figure B.3.: Tree illustrating some of the possible actions for the return of f

Static constraints require that $!f$ can only be activated if it has a sibling $?lock_f$, ensuring that its guard and constraints are true. In addition, $?lock_f$ acts as a lock disallowing any action other than the activation of the sibling $!f$. This is ensured by the following properties:

- $?lock_f$ and $!return$ do not occur simultaneously. (As in the simulation above, we require that each forest returned by a function contains a function call $!return$.)
- The argument query of $lock_f$ contains a query that returns the tag *error* if there is any activated function call $?lock_g$ in the instance. The static constraints forbid the occurrence of *error* in the instance.
- The initial workspace of each internal function is augmented with an internal function call $!activated$ (that returns the empty answer). Static constraints disallow the simultaneous occurrence of $?lock_f$ and $!activated$.

□

In summary, we have shown that

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

and $\text{GAXML}^{no\text{-}ret} \hookrightarrow_{(id,\pi)}$ BAXML. By Lemma 5.5 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 $\mathcal{V}_1 = \mathcal{V}_2 = id$ and $\mathcal{V} = \mathcal{V}_3 = \pi$. Since $\pi = id \circ \pi$ we have that GAXML $\hookrightarrow_{(\mathcal{V}_1,\mathcal{V}_2\circ\mathcal{V})}$ $\text{GAXML}^{no\text{-}ret}$ and $\text{GAXML}^{no\text{-}ret} \hookrightarrow_{(\mathcal{V}_2,\mathcal{V}_3)}$ BAXML. By Lemma 5.5, 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.

B.5.2. 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 5.5 implies that AAXML^{sib} can be simulated by BAXML.

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 automaton A. As described in the proof of Theorem 5.9, 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 $\varphi(next)$. The formula $\varphi(next)$ is the disjunction $\vee_{q,q'}\psi(next)(q,q')$, where q and q' are states of A, and $\psi(next)(q,q')$ 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 $\varphi(next)$ is not directly expressible as a static constraint in S', because these are Boolean combination of independent patterns, whereas $\varphi(next)$ uses parameterized patterns sharing free variables. To overcome this gap, some pre-processing is needed for each transition. Specifically, for a formula $\varphi(next)$ with free variables $\overline{X}$, candidate valuations for $\overline{X}$ are generated and the patterns in $\varphi(next)$ 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 $\varphi(next)$ to evaluation of a Boolean combination of independent patterns, so a static constraint of S'. In the following, we will use for simplicity $\varphi(next)$ as a static constraint, bearing in mind that its evaluation requires the above pre-processing phase.

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. But this is not sufficient. For some technical reasons, the queries cannot be rewritten in the same language. For example, $*//\$x$ is rewritten into the disjunction $*[real]/\$x \vee *[real]// * [real]/\$x$. Remark that disjunction cannot be expressed by a single query. To evaluate a disjunction, a sequence of internal functions, one for each disjunct in each query is called first. Each function simulates the evaluation of a disjunct and keeps the possible valuations. These valuations are used in the query at the end (details are omitted).

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 $\gamma_r(f)$ a function *check-return-guard*$_f$. The call guard of this function is $\gamma_r(f)$. The new local return guard of f simply checks that *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 $\gamma_r(f)$ 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 $S|A$. 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.

B.5.2.1. Activation of a function call

The activation of an internal function $!f$ is controlled using a sibling function $!lock_f$. 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 $!lock_f$ 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 $?lock_f$, (for some f) in an instance, i.e. at most one lock. This is enforced by the guard of $lock_f$. Moreover, no active function call can return its answer while $?lock_f$, !*activate-f*, or ?*activate-f* occur. As described in the proof of Theorem 5.12 , it is easy to ensure that every occurrence of $!f$ is always accompanied by a sibling $!lock_f$ 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 $!lock_f$ cannot be called more than once between two valid transitions. It is activated during the maintenance phase and returns $!lock_f$ (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 B.4 summarizes the possible sequences of activations in the simulation of an internal call of f. The role of the function $w_{f,a}$ will be explained shortly. The nodes represents the functions siblings of the call f The possible sequences for an external call are the same except the function $w_{f,a}$ is replaced by *certificate*$_{f,a}$.

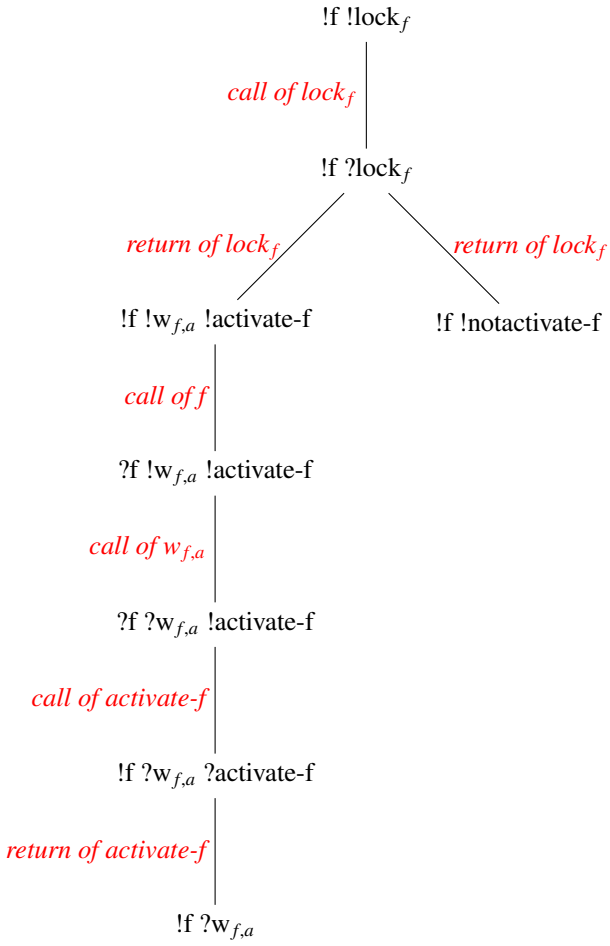


Figure B.4.: The tree some of the actions for the simulation of the activation of the call of f

B.5.2.2. 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 $!lock_w$ 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 $check_{f,a}$ 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 $check_{f,a}$ 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 $check_{f,a}$ 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). 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 $\varphi(next)$ has to be rewritten into a disjunction of formulas, each of which decomposes the patterns into a part that applies to the workspace of $check_{f,a}$ (mimicking the subtree rooted at the parent of the call $?check_{f,a}$, 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 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 the call $?check_{f,a}$ returns then the function calls $!w_{f,a}$ and $!reinitialize$. 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 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 $!check_{f,a}$ 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 $!eq_{(?g=m)}$ 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 B.5 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 $\text{check}_{f,a}$ when it is in the simulation.

B.5.2.3. 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 5.10). 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 $\textit{certificate}_{f,a}$ such that:

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

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

Note that, even if $?f$ can return, $?\textit{certificate}_{f,a}$ does not necessarily return, unless the return of $?f$ is the only possible next transition. Otherwise, the cleaning stage may be reached without a

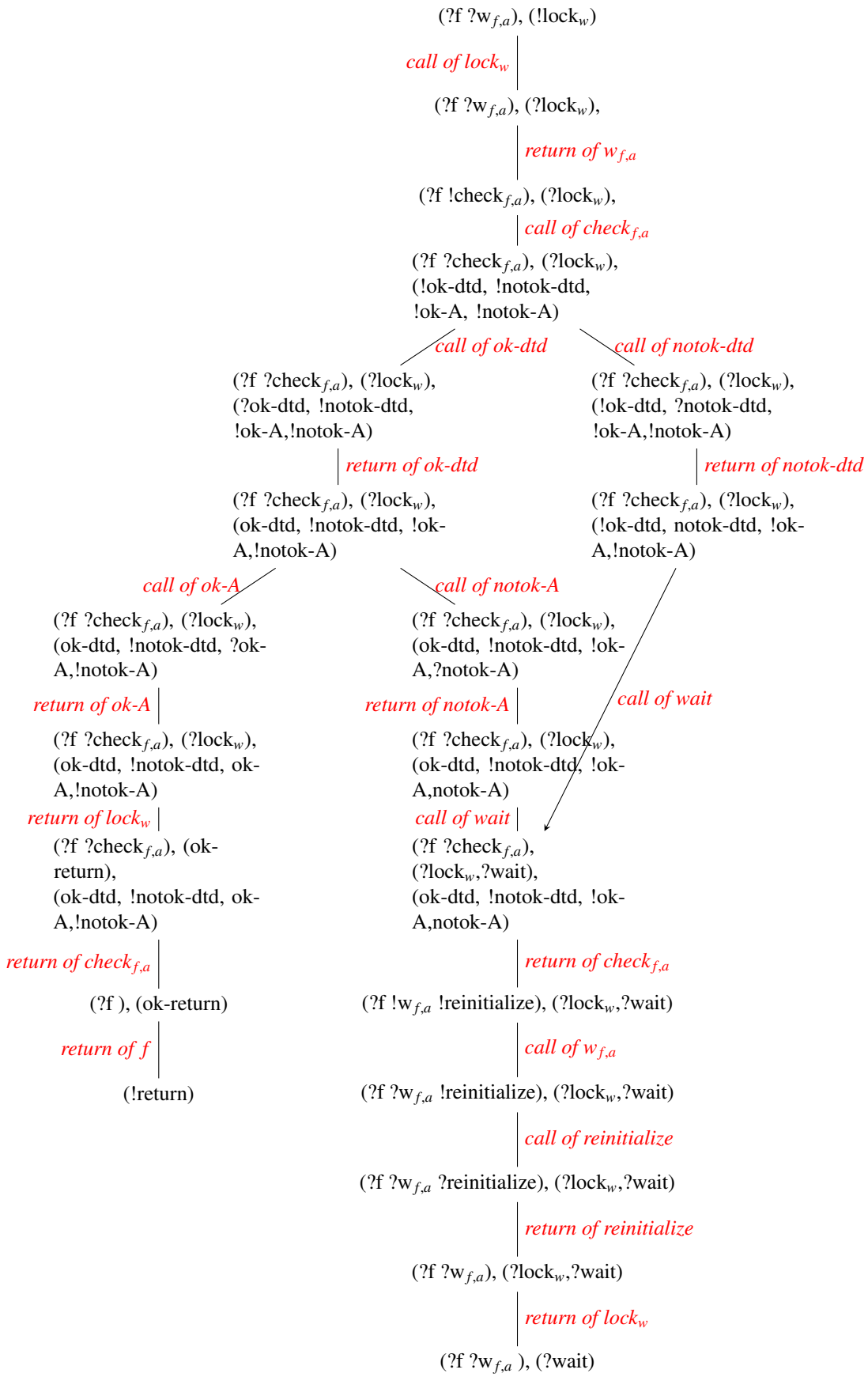


Figure B.5.: The tree of some of the actions for the simulation of the return of the internal call of f

return of ?$certificate_{f,a}$ or ?f, by simulating some other transition. If ?$certificate_{f,a}$ 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 $certificate_{f,a}$ uses in its workspace an external control function $fake_f$. The workspace also contains additional information so that ?$fake_f$ 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 $\varphi(next)$ 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 ?$certificate_{f,a}$ that are equivalent to those on the return of ?f (the DTD and valid transition in A). This ensures that ?$fake_f$ may return iff ?f may return. If ?$fake_f$ returns, then ?$certificate_{f,a}$ returns the flag !$return_f$ as desired. To prevent multiple returns to ?$fake_f$ at different locations in the document, the answer to ?$fake_f$ contains a flag !*return-fake-f* that is not allowed to appear twice in the document. To ensure this, the workspace of ?$certificate_{f,a}$ 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.

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.

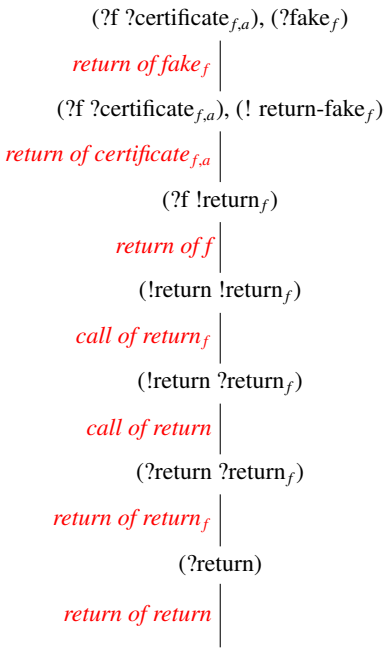


Figure B.6.: The tree of some of the actions for the simulation of the return of the call of the external function f

Figure B.6 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.

B.5.3. Simulation of TAXML^{sib} by BAXML

This follows from the simulation of AAXML^{sib} by GAXML and from Theorems 5.13, noting that the simulation of TAXML by AAXML does not introduce sibling calls to the same external function.

B.6. Proof of Theorem 5.13

B.6.1. Simulation of AAXML by 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 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. Recall that GAXML could not simulate AAXML with sibling calls to the same external function. We outline how this is done by TAXML.

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 $\varphi(next)$, whenever !*safe-r*$_f$ occurs (note that $\varphi(next)$ can be used directly in the Past-Tree-LTL formula).

B.6.2. Simulation of TAXML by AAXML

We describe the simulation of TAXML by AAXML. We first consider a variant of AAXML denoted AAXML* and show how TAXML can be simulated by AAXML*. We then show how AAXML* can be simulated by AAXML. In particular, it will follow from the constructions that

$$\text{TAXML}^{sib} \hookrightarrow_{(id,\pi)} \text{AAXML}^{sib}$$

completing the proof of Theorem 5.12.

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.

B.6.2.1. 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 96] 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 recall 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.

B.6.2.2. 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 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. The most delicate part of the simulation concerns (iii), i.e. the difference in the active domain semantics for the two models. There are two aspects to be dealt with. First, state formulas of A* are evaluated on the active domain of the entire prerun leading to the current state. This can be easily simulated in A by always propagating in a transition all values from the current to the next instance. The second, trickier aspect involves the new values introduced in transitions by external function calls. Because of the semantics, these have to be also taken 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 symbol, representing an arbitrary value outside the current active domain. Consider a valuation ν of the state variables $\overline{X}$ into the current active domain augmented with @. We can define satisfaction of a tree pattern $P(\nu(\overline{Y}))$ in a BAXML instance, where $\overline{Y} \subseteq \overline{X}$, as follows: if $\nu(\overline{Y})$ does not contain @, 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 can now be extended to keep states together with valuations including @. When new values are introduced, this provides sufficient information to determine the allowed transitions.

B.7. Proof of Theorem 5.14

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}), $0 \leqslant j \leqslant m$. 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 5.13) and AAXML cannot be simulated by GAXML. The difficulty is the same as in the above proof.

B.8. Proof of Theorem 5.22

We show that $\mathcal{TA}$ can be simulated by GAXML (this suffices since BAXML can simulate GAXML, see Theorem 5.12). We sketch the simulation for $\mathcal{TA}$ 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 dicussed in [Deutsch 09], 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 5.6). 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.

B.8.1. 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 to arbitrary arity). The relation P is represented by a subtree with root labeled P, satisfying the DTD

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

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 $!in\text{-}P$ and $!not\text{-}in\text{-}P$. More precisely, its initial workspace is of the following shape (the role of $!\bar{p}$ will be explained shortly):

$$\begin{array}{lll} a_{check_P} & \rightarrow & |!\bar{p}| = 1 \wedge |val| \geqslant 0 \\ val & \rightarrow & |dom| = 1 \wedge |!in\text{-}P| = 1 \wedge |!not\text{-}in\text{-}P| = 1 \end{array}$$

The functions *in-P* and *not-in-P* are internal. The call guard of *in-P* 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. Both can be easily done using relative patterns. The role of the function $!\bar{p}$ is dual. First, its guard ensures that for each value, one of its siblings $!in\text{-}P$ or $!not\text{-}in\text{-}P$ has been called. To this end, its guard forbids the presence of two siblings $!in\text{-}P$ and $!not\text{-}in\text{-}P$. Second, its argument query computes the complement of P, by collecting the values with a sibling $?not\text{-}in\text{-}P$.

B.8.2. 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.

B.9. Proof of Theorem 5.23

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 B.1. Specifically, each tuple is adorned with one function $!f_\tau$ 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 $\rho = init.e.g.(block)^\omega$ (we ignore the constant state). Indeed, since no function $!f_\tau$ 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 $T_\varnothing$ be the subtree in $\mathcal{T}_{init}$ corresponding to the empty database. From the above it follows that $V_S(T_{!e}) \sim V_\Gamma(T_\varnothing)$.

Recall that $\Delta \not\models f$ iff $V_S(T_{!e})$ contains a path from the root labeled $e.g.block$. Since $V_S(T_{!e}) \sim V_\Gamma(T_\varnothing)$, this happens iff $V_\Gamma(T_\varnothing)$ contains a path from the root labeled $\varepsilon^*.e.\varepsilon^*.g.\varepsilon^*.block$. By definition of $\sim$, since $V_S(T_{!e})$ has no infinite branches of ε-transitions (in fact no ε-transitions at all), $V_\Gamma(T_\varnothing)$ may not have infinite branches of ε-transitions. Also note that $T_\varnothing$ 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 $V_\Gamma(T_\varnothing)$ (modulo isomorphism) until the first 3 non-ε transitions occur along all branches. This allows deciding if a path labeled $\varepsilon^*.e.\varepsilon^*.g.\varepsilon^*.block$ starting from the root exists in $V_\Gamma(T_\varnothing)$, and provides a procedure for testing whether $\Delta \models f$.

Appendix C.

Résumé de la thèse en français

C.1. Introduction

L'un des principaux problèmes que les applications Webs doivent gérer aujourd'hui est l'évolutivité des données. Dans cette thèse, nous considérons ce problème et plus précisément l'évolution des documents actifs, une abstraction des documents écrits en Active XML [Abiteboul 08a] (AXML). Les documents actifs sont documents XML pouvant évoluer grâce à l'activation d'appele de services Web. Ce formalisme a déjà été utilisé dans le cadre de la gestion d'information distribuée. Les principales contributions de cette thèse sont l'étude théorique de différentes notions pour l'implémentation de deux systèmes gérant des applications manipulant des flux de données et des applications de type workflow. Dans un premier temps, nous étudions des notions reliées à la maintenance de vue sur des documents actifs. Ces notions sont utilisées dans l'implémentation d'un processeur de flux de données appelé Axlog widget manipulant des flux à travers un document actif. La deuxième contribution porte sur l'expressivité de différents formalismes pour contraindre le séquencement des activations d'un document actif. Cette étude a été motivée par l'implémentation d'un système gérant des workflows focalisés sur les données utilisant les documents actifs, appelé Axart. Dans une première partie, nous étudions les applications gérant des flux. Le Web possède un nombre très important de sources de flux de données exprimées en XML comme les fils de blogues ou les fils de dépêche. De plus en plus, les pages Webs font simplement de l'édition d'agrégats de ces flux. Au cœur de ces pages se trouve un système de gestion de flux. Nous présentons un modèle formel appelé Axlog qui calcule des requêtes simples sur les flux. Notre approche se fonde sur la maintenance de vues sur les documents actifs. La contribution principale autour de ce sujet, publiée dans [3], est l'étude de deux notions théoriques appelées satisfiabilité et pertinence. Un algorithme pour maintenir une vue sur un document incluant des appels à des flux est brièvement présenté [4]. Cet algorithme s'articule autour des notions théoriques présentées précédemment. Ces notions permettent de combiner ensemble différentes techniques relatives aux bases de données : optimisation de l'évaluation d'une requête écrite en datalog, techniques pour la maintenance de vues, gestion de calculs de flux XML et enfin techniques de sélections pour les flux XML. Le modèle Axlog est implémenté par le système P2PMonitor, présenté dans [9]. P2PMonitor est un système distribué qui a pour finalité la surveillance d'autres systèmes distribués. La surveillance faite par P2PMonitor est effectuée par le calcul de requêtes sur des flux de données XML. Le système P2PMonitor et l'algorithme de maintenance de vue ont été présenté en détail dans la thèse de Bogdan Marinoiu [Marinoiu 09]. Dans une deuxième partie, nous abordons le problème du séquencement des actions intra et inter applications. Les sites Webs d'e-commerce sont un exemple intéressant d'applications où l'ordonnancement est crucial. Les interactions entre utilisateurs et applications sont contraintes afin d'être conformes au workflow. Plusieurs langages de description de workflows ont été introduits, comme BPEL. Cependant, grand nombres de ces langages se concentrent sur la description procédurale des possibles séquencements des actions sans prendre

vraiment en compte d'autres aspects comme les données utilisées dans l'application.

De nouveaux types de langages de description de workflows plus focalisés sur les données, appelés «data-centric workflow» ont été récement proposés. Nous présentons dans cette thèse le modèle d'AXML Artifact, [5], influencé par le modèle de Business Artifact, introduit par IBM. Notre principale contribution, [2], dans cette partie est l'étude et la comparaison de différentes manières d'exprimer l'ordonnancement des actions en se basant sur des paradigmes différents incluant automates, pré-et post-conditions pour l'activation d'une action ou la logique temporelle. Nous décrivons brièvement un système, AXART [8], implémentant une partie du modèle AXML Artifact.

La thèse est organisée en deux parties. La première traite des applications gèrant les flux et la deuxième des workflows. Chaque partie est organisée de la même manière. Après une présentation de la partie, nous discutons des travaux similaires. Nous présentons le modèle et l'étude théorique pour le type d'applications que nous considérons. Pour conclure, nous discutons de l'implémentation du modèle présenté précédemment.

C.2. Résumé de la partie I : Maintenance de vues sur les documents actifs

Les échanges de flux d'information est un des mécanismes prédominants des applications Webs. Ces mécanismes sont particulièrement présent dans les mashups systèmes [Ennals 07] ou systèmes distribués dédiés à la surveillance [Abiteboul 07]. La gestion de ces flux de données est un ambitieux problème qui a déjà engendré un très grand nombre de résultats voir Web 2.0 [O'Reilly]. Nous présentons un nouveau modèle se fondant sur datalog et Active XML technologies appelé Axlog. Il capture les interactions entre applications Webs et montre comment celles-ci peuvent implémentées efficacement. Un Axlog widget utilise un document actif interagissant avec le reste du monde au travers des flux de notifications. Leurs flux entrants notifient les modifications du document (dans l'esprit des fils RSS) tandis que leur flux sortant est définie par une requête posée sur le document. Plus précisément un flux sortant représente la liste des mises-à-jour nécessaires pour maintenir la vue définie par la requête. Les requêtes que nous considérons dans cette thèse sont des requêtes de motif d'arbre avec jointures sur les données (et un motif pour produire un résultat sous format XML). Notre modèle de données et les requêtes associées peuvent inclure une dimension temporelle essentielle dans les problèmes de surveillance. La clé du support d'un Axlog Widget est la maintenance de la vue correspondant à la requête posée sur le document actif. Pour cela, nous réutilisons un ensemble de technologies : des techniques d'optimisation de requêtes datalog comme MagicSet, des techniques efficaces pour la maintenance de requêtes datalog comme Differential et des techniques efficaces de filtres sur des flux XML. Le cœur des optimisations de notre algorithme est construit autour de deux techniques fondamentales : la pertinence (différente de celle introduite dans Magic Set) et la satisfiabilité d'une requête sur un document actif. Dans un premier temps nous introduisons le cœur du modèle où les mises-à jour sont simplement des insertions et les requêtes sont des motifs d'arbres. Nous présentons et étudions les deux concepts fondamentaux que sont la satisfiabilité et la pertinence. Un fait est satisfiable pour un document actif et une requête, s'il a une chance de devenir vrai par une suite de mise-a-jour du document actif. Pour un document actif et une requête donnés, une souscription à un flux du document est pertinente si les mises-à-jour apportées par le flux peuvent avoir un impact sur le résultat de la requête. Nous analysons la complexité de la satifiabilité pour le cœur du modèle ainsi que pour des extensions : un document actif peut être associé à un type, les mises-à-jour peuvent inclure

également des supressions, les requêtes peuvent inclure des négations et documents et requêtes peuvent inclure une dimension temporelle. Les Axlog widgets sont les composants essentiels du système P2PMonitor. Nous le présentons briêvement ainsi que l'intégration des Axlog widgets dans celui-ci. Nous expliquons sans détailler comment la satisfiabilité et la relevance sont utilisées dans l'algorithme pour la maintenance d'une vue. Pour résumer, les Axlogs widgets peuvent être utilisés pour supporter un grand nombre de tâches dans des environnements distribués comme la gestion de flux. L'utilisation des Axlog Widgets a été démontrée dans [9] dans le cadre de la surveillance d'une application de construit de produit [Kapuscinski 04]. Les Axlog widgets sont au cœur d'une seconde version du système P2PMonitor [Abiteboul 07]. L'implémentation est basée sur un algorithme efficace de maintenance de vue [4]. Cet algorithme utilise notamment les notions théorique de satisfiabilité et pertinence étudiés dans [3].

C.2.1. Résumé du Chapitre 2 : Axlog Model, satisfiabilité et pertinence

Dans ce chapitre, nous introduisons le cœur du modèle Axlog, qui est un document actif avec une requête définissant une vue dessus. Un document actif est un arbre avec sémantique ensembliste pour les sous-arbres d'un arbre et interagissant avec le reste du monde avec des flux de mises-à-jour. Les requêtes sont des motifs d'arbre avec des jointures. Le résultat d'une requête sont les n-uplets représentant les possibles valeurs que peuvent prendre certaines des variables apparaissant dans le motif d'arbre. Ces n-uplets peuvent être édités sous la forme d'arbres en utilisant un formulaire. Un document actif associé avec une requête est ce que nous appelons une Axlog widget. Le terme d'Axlog vient du mariage d'Active XMl (AXML en acronyme) [Abiteboul 08a] et datalog. Les flux entrants d'une Axlog widget définissent les mises-à-jour du document actif. Dans la plupart des résultats, les mises-à-jour sont des insertions. Nous étudions toutefois le cas où les mises-à-jour sont des suppressions. Le flux sortant est défini à partir de la requête de l'Axlog widget. Plus précisément, il représente la liste des mises-à-jour nécessaires pour maintenir la vue définie par la requête. La majeure contribution de cette partie est l'étude des notions pour les documents actifs de la satisfiabilité et de la pertinence. Tout d'abord, un fait est satisfiable pour un document actif et une requête si ce fait peut apparaître dans la réponse de la requête à la suite de mises-à-jour du document actif. Dans le même esprit que l'évaluation de motifs d'arbres en utilisant un programme datalog [Gottlob 02], nous démontrons comment évaluer la satisfiabilité d'une requête (si il y a au moins un fait satisfiable) en utilisant un programme datalog. Il en découle que la satisfiabilité d'une requête est PTIME dans la taille du document actif. Remarquez que le nombre de faits satisfiables pour un document actif et une requête peut être infini. Nous utilisons une représentation finie en utilisant des n-uplets possédant des variables. Pour gérer ces n-uplets, appelés n-uplets généralisés, nous utilisons le langage de requêtes CQL [Kanellakis 95]. La satisfiabilité est aussi étudiée pour des extensions du modèle. Premièrement, les documents actifs et les données des mises-à-jours sont typées [Comon 97, DTD , WSDL]. Comme nous utilisons une sémantique ensembliste pour les sous-arbres d'un nœud et des arbres non ordonnés, nous adaptons les DTDs pour ignorer l'ordre des nœuds voisins. Nous démontrons que la satisfiabilité d'une requête pour un document actif typé à la même complexité. Toutefois l'algorithme que nous utilisons n'est pas un simple programme datalog. D'un autre côté, nous étudions plusieurs extensions entrainant la non monotonicité de la satisfiabilité de la requête pour un document actif telles que suppression, terminaison d'un flux, négation de sous-requêtes. En particulier, la négation dans les requêtes entrainent l'indécidabilité de la satisfiabilité. Pour finir, nous introduisons le temps dans les documents et les requêtes. Le temps est une importante notion pour la surveillance. Nous étendons le programme utilisé dans le modèle de base pour tenir compte des contraintes temporelles.

La deuxième notion que nous étudions est la pertinence d'une souscription à un flux. La souscription à un flux est pertinente si les données apportées par cette souscription peuvent avoir un impact sur le résultat de la requête. Une notion de pertinence plus faible a déjà étudiée d'appels de fonctions dans [Abiteboul 04a]. La pertinence dans le cas de base est PTIME dans le document actif. Hélas, la complexité combinée est élevée et l'algorithme ayant une complexité polynomiale possède des constantes trop élevée en pratique, Nous introduisons une notion de pertinence entre notre notion de pertinence et la pertinence proposée dans [Abiteboul 04a], appelée axlog–pertinence. C'est cette notion qui est utilisée dans l'implémentation.

C.2.2. Résumé du chapitre 3 : Implémentation du modèle d'Axlog

Les travaux faits autour d'Axlog ont été premièrement motivés par le développement du système P2PMonitor, un système pour la surveillance des applications pair à pair introduit dans [Abiteboul 07] et développé dans [Marinoiu 09]. À partir de notre algorithme de maintenance de vues, le système P2PMonitor supporte maintenant les Axlog widgets [4]. Leur utilisation a été démontrée dans [9] dans le cadre de la surveillance d'un système simulant sur la chaine d'assemblage Dell. Le problème principal pour l'implémentation des Axlog widgets est le calcul efficace de la production du flux de sortie, qui est un problème de maintenance de vue. Nous avons développé un tel algorithme qui est aussi présenté dans [4]. Cet algorithme combine des techniques de bases données comme MagicSet [Beeri 91] et Differential [Blakeley 86b] ainsi des techniques de filtrage de flux [Diao 02] qui n'avaient pas jamais été utilisées ensemble. Cette combinaison est possible dans notre cadre grâce à l'utilisation des notions que sont la satisfiabilité et la pertinence. Pour donner un bref aperçu de ce que fait l'algorithme, MagicSet permet le calcul que des faits pertinents pour l'évaluation de la requête actuelle. Cela peut entraîner le calcul d'un très grand nombre de faits à l'arrivée d'une mise-à-jour qui aurait pu être évité car ils étaient calculables avant cette mise-à-jour. Nous relaxons l'algorithme de MagicSet en autorisant le calcul des faits satisfiables. Grâce à notre algorithme du calcul de pertinence présenté dans le travail théorique, nous sommes en mesure d'indiquer quelles requêtes doivent être satisfaites par les données arrivant d'une souscription d'un flux pour avoir un impact sur le résultat de la requête. Grâce à ces connaissances, il est possible de filtrer les données des flux avant leur intégration au programme de maintenance de vue (sauvant du temps de calcul) et si possible à la source même (sauvant des communications).

C.3. Résumé de la partie II : le modèle AXML Artefact

L'évolution de données partagées est le centre des nombreuses activités humaines constituées de tâches qui sont coordonnées au travers d'un worklflow. Les langages de description de workflow se sont globalement focalisés sur des descriptions procédurales des actions ignorant les aspects liés aux données. Depuis récemment, il y a une prolifération des langages, appelés workflow centré-données, en réponse aux besoins croissants de modélisation des applications manipulant massivement des données. E-commerce systèmes, processus de business d'entreprise, processus médicaux et scientifiques workflows sont des exemples primordiaux nécessitant des workflows centrés données. Dans le cadre d'une approche de workflow centré données, nous proposons un nouveau modèle de description de workflow centré données, appelé AXML Artefact. Dans un premier temps, nous présentons dans un premier temps le cœur du modèle AXML Artefact, appelé BAXML pour capturer la gestion de l'activité d'un document actif. Ce modèle utilise des documents actifs qui dont l'évolution est seulement contraint pour des contraintes statiques. Un des

plus grands challenges discuté de la modélisation des workflows centré données est la spécification des contraintes de workflow. Nous étudions le problème de comparer l'expressivité de plusieurs mécanismes de spécification de workflow en utilisant une notion de vue pour un workflow. Les vues permettent de comparer des systèmes décrits dans des langages très largement différents en associant aux différents systèmes une représentation commune capturant les notions observables pertinentes pour la comparaison. Nous comparons l'expressivité de plusieurs langages de workflow incluant automate, contraintes temporelles et pré- et post- conditions associés au BAXML comme le modèle de données sous-jacent. Un résultat surprenant montre que les contraintes statiques ont un pouvoir d'expression considérable permettant de simuler de mécanismes de contrôle plus complexe. De plus, nous affirmons que notre modèle capture l'essentiel des caractéristiques business artefacts comme décrits informellement dans [Nigam 03a] ou discuter dans [Hull 08]. Pour motiver cette affirmation, nous comparons l'expressivité du modèle BAXML avec le modèle d'AXML artefact introduit dans [5]. Dans un deuxième temps, nous étendons le modèle AXML Artefact avec différents principes: hiérarchie d'artefacts, contrôle d'accès, dynamique modification des comportements possibles du système et distribution dans le système. Les notions de hiérarchies et de distribution ont été présenté [5] et [Hélouët 10]. La distribution est un problème difficile pour l'implémentation [Marinoiu 09] et la vérification [Hélouët 10]. Cette extension n'est pas étudiée dans cette thèse. Les autres extensions sont nouvelles. Nous illustrons ces notions à l'aide de deux exemples venant des applications de type Business : un exemple basé sur la gestion d'un casting [Wikipedia] et un exemple basé sur la chaîne d'assemblage Dell [Kapuscinski 04]. La gestion du casting illustres les trois premières notion tandis que la chaîne d'assemblage Dell illustre la distribution. Pour conclure, nous présentons le système AXART supportant la majeure partie du modèle AXML Artefact, avec les extensions. Ce système se focalise sur les interactions humaines gérées par un système centralisé. Cela implique que la distribution n'est pas gérée par le système. Pour résumer, nous présentons un langage pour décrire des workflows centrés données appelé AXML Artefact. Il est basé sur les documents actifs. La contribution majeure de cette partie, publiée dans [2], est une comparaison de l'expressivité de différents formalismes pour le contrôle de l'évolution d'un document actif (BAXML). Nous avons implémenté un prototype d'une grande partie d'AXML Artefact, démontré en [8]. La distribution qui n'est pas supporté par le prototype AXART, est simplement présenté dans cette thèse mais est pleinement développé dans [Marinoiu 09].

C.3.1. Résumé du chapitre 5 : Comparaison de l'expressivité de langages de spécifications de workflows: un problème de vues

L'évolution de données partagées est le centre d'un grand nombre d'activités humaines. La notion de business artefact [Nigam 03a] a été proposée pour spécifier de telles évolutions. L'idée principale est de capture le flux de contrôle de workflow de l'application mais aussi l'évolution des données. En partant des documents actifs, nous proposons un nouveau artefact modèle, appelé AXML Artifact modèle. Le séquencage des documents actifs peuvent être spécifiés par différents mécanismes comme les automates, les pré- et post-conditions ou des contraintes temporelles. Le but principale est de comparer ces différents mécanismes. Toutefois, il est extrêmement difficile de comparer ces langages de spécification au vue de la diversité des formalismes et du manque de référence commune. Dans ce chapitre, nous développons un adaptive cadre pour comparer les langages de spécification de workflows où les aspects pertinents, qui doivent être priss en cause, sont définis au travers de vues. Nous utilisons ce cadre pour comparer les langages présentés précédemment. Considérons le système qui évolue dans le temps comme le résultat de calculs internes ou d'interactions avec le reste du monde. Fondamentalement, une spécification de workflow

impose des contraintes sur son évolution. Il existe un grand nombre d'approches pour spécifier de telles contraintes. La plus populaire des approches semble consister à spécifier un ensemble d'états abstraits du système et d'imposer des contraintes sur les transitions possibles entre états, comme dans BPEL [BPEL]. D'autre part, une approche plus déclarative est de définir un ensemble de tâches équipées avec des pré- et post- conditions, tel que le modèle d'IBM, Business artefact. Un système d'artefact peut aussi imposer des contraintes par des formules temporelles sur l'histoire de l'exécution [Hull]. Nous affirmons que les vues sur les workflows est très utile pour comparer des workflows mais aussi que les vues permettent de décrire plusieurs explications d'un même workflow. Par exemple, les vues peuvent être utilisé pour des interfaces aux clients de différentes classes d'actionnaires pour de raisons de présentation ou de sécurité. Les interactions et contrats sur les workflows peuvent être aussi spécifiés par des vues. La création de worklows complexes est faite naturellement en procédant par rafinement par vues abstraites. Les vues peuvent être également utilisées à l'exécution pour la surveillance, la détection d'erreurs, le diagnostique ou capturer les souscriptions de requêtes continues. Le mécanisme d'abstraction fournis par les vues est également essentiel dans l'analyse statique et la vérification. Dépendant des besoins nécessaires, une vue de workflow peut retenir des informations concernant des états abstraits du système, de son évolution, comme sur certains évènements et leur séquences, ou sur l'entière histoire du système jusqu'à maintenant ou la combinaison de ces aspects. Même si cela n'est pas fait explicitement, une vue est souvent le point de départ de la création d'un workflow.

Pour voir comment cela marche, considérons un workflow W spécifié par des tâches et des pré-/post-conditions et un autre workflow W_0 spécifié par un système à transitions, les deux pertinents pour la même application. Un moyen pour rendre les deux workflows comparable est de définir une vue sur W comme un système à transitions compatible avec W_0. Cela peut être faire en définissant état à partir de requêtes sur l'état courrant et les transitions induites par les tâches. Pour rendre la comparaison significative, la vue doit retenir dans les états les informations pertinentes pour la sémantique de l'application restructurée pour être compatible avec la représentation utilisée dans W_0. Plus généralement, les vues peuvent être utilisées pour des langages totalement différents. Nous formalisons la générale notion de vue et introduisons une forme de bisimulation sur les vues pour capturer le fait qu'un worflow simule un autre. Dans notre formel développement, nous étudions principalement les documents actifs (AXML [Abiteboul 08a]) qui permet une intégration continue de processus et de données. Pour décrire, l'évolution du système en l'absence de contraintes de type workflow, nous définissons le modèle de base appelé Basic AXML (BAXML). BAXML documents sont des documents actifs qui sont adaptés dans le contexte des workflows. Le document évolue suivant le résultat d'appels de fonctions qui initient des sous-tâches et qui retourne des sous-arbres . Les fonctions peuvent être internes ou externes, suivant la modélisation des interactions avec l'environnement. La figure C.1 montre un document actif. Les documents sont sujets à des contraintes statiques définies par une DTD et une combinaison booléenne de motifs d'arbres. Remarquez que cela implique déjà une certaine forme de contrôle avec le flux d'exécution. En effet, un appel de fonction peut être activé ou peut retourner un résultat que si le document résultant ne viole pas les contraintes statiques. En effet, nous verrons que cela produit déjà un très grand pouvoir pour exprimer des contraintes de type workflow. BAXML offre un naturel modèle pour spécifier les exécutions d'un système dans lequel les tâches correspondent à des documents évoluant et les appels de services sont des requêtes retournant des sous- tâches. À partir de ce modèle de basem nous considérons trois moyens de spécifier augmentant BAXML avec des contrôles explicites, correspondant aux trois importants workflows paradigmes :

- Automates : les automates sont non déterministes systèmes transitionnels finis. Des requêtes

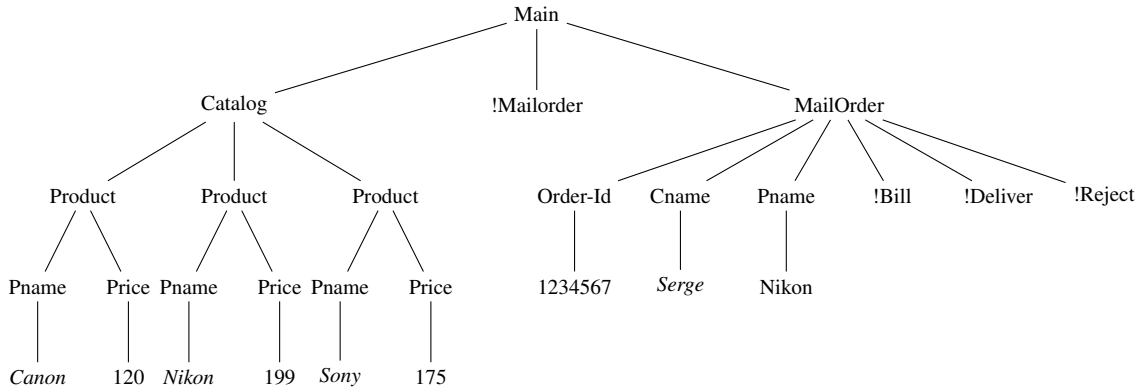


Figure C.1.: A BAXML document.

sont associées à leurs états. Les requêtes sont des formules de motifs d'arbre avec des variables libres apparaissant dans la paramétrisation des états. Une transition dans un état peut être effectuée si la formule associée au nouvel état est vrai dans le document résultant. De plus, l'automate peut contraindre les valeurs des paramètres de deux consécutives états.

- Gardes : les gardes sont des pré-conditions controllant l'activation des appels de services et le reoutr de leur réponses. Ce mécanisme de contrôle a été introduit dans [Abiteboul 09], dont les résultats concernent la vérification de propriétés temporelles sur de tels systèmes.

- Propriétés temporelles : elles sont exprimées dans une logic utilisant des motifs d'arbres et Past-LTL opérateurs. Une formule temporelle contraint l'évolution d'un document en utilisant l'histoire de l'exécution actuelle.

Bien que ceci est présenté dans le contexte de BAXML, ces extensions capturent les aspects essentiels des paradigmes de spécification indépendemment du modèle de données sous-jacent.

Notre résultats principaux concernent le pouvoir d'expressivité de BAXML et des différentes extensions liées aux langages de spécifications de langages. Quand nous imposons que deux systèmes aient exactement les mêmes exécutions, les trois extensions sont incomparables. Des résultats deviennent plus intéressants quand nous relachons une notion plus permissive d'équivalence dans laquelle une vue peut cacher des portions de donńées et des activités liées aux fonctions permettant plus de souplesses dans la simulation. Surprenamment, nous démontrons que BAXML est capable de simuler largement les trois mécanismes de spécifications basés sur les gardes, les automates ou les propriétés temporelles. Cela indique la puissance impressionante des contraintes statiques pour simuler des mécanimses apparemment plus puissants. Bien sûr, les spécifications comme les gardes, les automates et propriétés temporelles sont plus lisibles que leur équivalentes spécifications dans BAXML en utilisant des fontions non observables et des contraintes statiques. Les résultats présentés montrent l'utilité de montrer une abstraction du worklflow comme une contrainte sur les exécutions du système sous-jacent, découplé des approches spécifiques pour définir les contraintes. Il est aussi démontré que l'utilité des vues dna sla comparaisons des workflows et des langages de workflows. Bien que les langages décrits ici sont formalisés dans le contexte d'AXML, nous croyons que les résultats montrent la généralisation de l'approche sur d'autres formalisme que celui présenté ici. En particulier, les preuves donnent des idées générales de comment les

différents langages peuvent se simuler l'un l'autre. Après les résultats relatifs aux document actifs, nous considéron le modèle de Business artefact introduit par IBM qui utilisent des paradigmes différents : les données sont représentées sous un format relationnel et les services sont équipées de pré-/post-conditions en utilisant des formules du premier ordre. Grâce encore une fois aux vues, nous comparons BAXML avec le modèle de Business artefact comme formaliser dans [Deutsch 09]. Nous prouvons que BAXML peut simuler les artefacts mais l'inverse est faux. Le premier résultat utilise des vues associant des arbres XML à des relations et des fonctions à des services. Pour le résultat négatif, nous utilisons des vues qui retiennent seulement la trace des appels de fonctions dans BAXML et des services dans le modèle de business artefact. Cela montre un très for résultat, il s'étend pour toute vue qui expose plus d'information que les traces d'appels de fonctions/services. Ces résultats montrent encore la flexibilité et l'intérêt des vues pour comparer des formalismes de workflows..

C.3.2. Résumé du Chapitre 6: Implémentation du modèle AXML Artefact

. Nous étendons le coeur du modèle d'AXML Artefact par plusieurs intéressantes notions, le contrôle des interactions avec des utilisateurs, la modification dynamique du workflow, la hiéarchisation des tâches et la distribution. La modification dynamique du workflow et la gestion du contrôle d'accès sont nouveaux comparer aux travaux antérieurs. Nous les illustrons dans un exemple pratique : la gestion de l'attribution de la distribution d'un film comme décrit dans [Wikipedia]. Ceci est repésentatif des interactions possibles entre plusieurs acteurs humains. L'exemple montre comment un standard workflow est modélisé grâce le modèle d'AXML artefact et comment les utilisateurs peuvent facilement exécuter le workflow en interagissant avec les artefacts en utilisant des formulaires et en modifiant les états associés aux artefacts. Les artefacts sont surveillés et des notifications peuvent être possiblement envoyés aux utilisateurs qui peuvent prendre les bonnes dès lors la décision appropriée. L'exemple illustre la modification dynamiques des workflows par deux mécanismes appelés spécialisation de workflows et relation de workflow qui permettent aux utilisateurs ayant les autorisations de modifier le workflow au cours de son exécution. Cela procure une flexibilité utile lors de la spécification d'un workflow avec notre modèle. Toutes les modifications sont contrôlées par règles d'accès. La distribution est illustrée par une modélisation de la chaine de distribution de Dell [Kapuscinski 04].

Un prototype, AXART du modèle d'AXML artefact est présenté. Ce prototype sert à gérer les interactions entre humains. Il implémente qu'un sous-ensemble du modèle étendu. Par exemple, les seules fonctions gérées sont les fonctions externes qui représentent les interactions avec les utilisateurs. L'idée principale du système est de gérer efficacement les requêtes évoluant dans les règles du workflow. Pour cela, le système maintient incrémentalement ces requêtes. Il utilise les techniques de surveillance développées dans la première partie telles que la maintenance de motifs d'arbres sur des document actifs en présence de souscriptions de flux générant que des ajouts d'arbres. AXART combine ces techniques avec des technique de sécurité sur le contrôle d'accès. Cela permet la gestion d'une collection raisonnale d'artefact munis de contrôle d'accès.

C.4. Conclusion

Nous étudions dans cette thèse comment les documents actifs peuvent être considérer comme un modèle de base pour la gestion d'information distribuée. Les études se sont focalisées sur deux types de systèmes : la gestion de flux de données et les worklows centrés données.

C.4.1. Gestion de flux : le modèle Axlog

Notre première contribution est l'étude des documents actifs dans le cadre de la gestion de flux. Pour cela un modèle, Axlog, a été proposé. Il est la juxtaposition d'un document actif souscrivant à des flux de données et d'une requête définissant un flux de sortie. Notre intérêt est de présenter un algorithme efficace pour maintenir la requête durant l'évolution du document actif. Pour cela nous introduisons deux notions : la satifiabilité et la pertinence. La faisabilité du calcul de ces deux notions est étudiẹe dans cette thèse.

Worklow centré données: le modèle AXML Artefact

Notre secondde contribution est d'étudier les documents actifs pour la modélisation d'application de workflows centrés données. Pour cela, nous proposons un modèle sous-jacent, BAXML fait de documents actifs et de contraintes statiques auquel des contraintes de workflows sont associées. La contribution majeure a été d'étudier l'expressivité de différents langages de contraintes en utilisant un nouveau formalisme basé sur les vues. Nous avons en parituclier montré la puissance des contraintes statiques pour simuler des formalismes qui semblaient plus expressifs.

C.4.2. Perspectives

Les workflows centrés données proposent de grands défis et il reste beaucoup de travail à faire. Nous proposons plusieurs approches.

La première est l'extension du système AXART pour gérer la distribution comme proposé dans le modèle général. Cela implique des problèmes pour la gestion des contrôles d'accès. Une première approche serait de poursuivre les approches proposées dans [Abiteboul r] et dans [Abiteboul f] et démontré dans [Antoine 11]. Une autre approche serait de faire de la vérification a posteriori des actions des différents utilisateurs. Cela impliquerait de développer des sigatures utiles dans notre cadre. Les deux autres points sont d'étudier la notion de vues proposées dans cette thèse pour définir des abstractions riches de systèmes pour permettre de définir des interfaces ou faire de la vérification de parties utiles du modèle.

Contents

Introduction 1

I. Maintenance of Views over Active Documents 3

1. Related Work 7
1.1. Stream processing systems . . . 7
1.1.1. Relational streams systems . . . 7
1.1.2. XML streams systems . . . 7
1.2. Query evaluation . . . 7
1.2.1. Datalog . . . 7
1.2.2. AXML . . . 8
1.3. View Maintenance . . . 8
1.3.1. Incremental View Maintenance . . . 8
1.3.2. Relevance . . . 8
1.4. Satisfiability and containment of queries . . . 9
1.4.1. Satisfiability of tree-pattern queries over static trees . . . 9
1.5. Incomplete information . . . 10
1.5.1. Incomplete relational databases . . . 10
1.5.2. Trees and incomplete information . . . 10
1.6. Type . . . 10
1.6.1. Unordered trees . . . 10
1.6.2. Reduced trees . . . 11

2. Satisfiability and relevance for queries over active documents 13
2.1. Introduction . . . 13
2.2. The model . . . 14
2.2.1. Definitions . . . 14
2.2.2. Axlog Systems . . . 17
2.3. Satisfiability . . . 18
2.4. Typed documents . . . 22
2.4.1. Schema definition . . . 22
2.4.2. Satisfiability for Axlog schemas . . . 23
2.5. Nonmonotonicity . . . 24
2.6. Relevance . . . 27

3. Axlog 31
3.1. Introduction . . . 31
3.2. Axlog at Work . . . 31
3.3. The View Maintenance Algorithm . . . 32

II. Data-centric Workflow Applications **37**

4. Related Work **41**
4.1. Workflow languages and models . . . 41
4.2. Workflows systems . . . 42
4.3. Formal studies about workflows . . . 42
4.3.1. Comparison of workflow languages . . . 42
4.3.2. Verification . . . 42

5. Comparing Workflow Specification Languages: A Matter of Views **45**
5.1. Introduction . . . 45
5.2. Views and Simulations . . . 47
5.3. The Basic AXML model . . . 50
5.4. Workflow Constraints . . . 56
5.5. Expressiveness . . . 60
5.6. BAXML and Tuple Artifacts . . . 62
5.6.1. The Tuple Artifact Model . . . 63
5.6.2. Comparison . . . 65

6. An implementation of the AXML artifact model: AXART **67**
6.1. Introduction . . . 67
6.2. Two motivating examples . . . 68
6.3. Extensions of the AXML Model . . . 71
6.4. The AXART System . . . 73
6.4.1. The implemented Model . . . 73
6.4.2. Algorithm . . . 74
6.4.3. The Architecture . . . 74
6.4.4. The User Interface . . . 75

Conclusion **77**
Stream Processing: the Axlog model . . . 77
Data centric workflow: the AXML Artifact model . . . 77
Perspectives . . . 78

III. References **79**

IV. Appendix **93**

A. Satisfiability and Relevance: Proofs **95**
A.1. Notations . . . 95
A.2. Satisfiability . . . 95
A.2.1. Proof of Theorem 2.6 . . . 95
A.2.2. Proof of Theorem 2.7 . . . 98
A.2.3. Systems of active documents . . . 100
A.3. Typed documents . . . 100

A.3.1. Reduced DTDs . . . 100
A.3.2. Reduced Axlog schemas . . . 108
A.3.3. Proofs of Theorems 2.11 and 2.12 for reduced Axlog schemas . . . 109
A.4. Nonmonotonicity . . . 116
A.4.1. Noninflationary documents: Theorem 2.13 . . . 116
A.4.2. Non monotonic queries . . . 119
A.5. Relevance . . . 122

B. Comparing Workflow Specification Languages: The Proofs 125
B.1. Proof of Theorem 5.7 . . . 125
B.2. Proof of Theorem 5.9 . . . 126
B.3. Proof of Theorem 5.10 . . . 127
B.4. Proof of Theorem 5.11 . . . 127
B.4.1. Lemma GAXML $\not\hookrightarrow_{(id,id)}$ AAXML . . . 128
B.4.2. Lemma GAXML $\not\hookrightarrow_{(id,id)}$ TAXML . . . 128
B.4.3. Lemma AAXML $\not\hookrightarrow_{(id,id)}$ GAXML . . . 129
B.4.4. Lemma TAXML $\not\hookrightarrow_{(id,id)}$ GAXML . . . 129
B.4.5. Lemma TAXML $\not\hookrightarrow_{(id,id)}$ AAXML . . . 129
B.4.6. Lemma AAXML $\not\hookrightarrow_{(id,id)}$ TAXML . . . 130
B.5. Proof of Theorem 5.12 . . . 130
B.5.1. Simulation of GAXML by BAXML . . . 130
B.5.2. Simulation of AAXML^{sib} by BAXML . . . 133
B.5.3. Simulation of TAXML^{sib} by BAXML . . . 141
B.6. Proof of Theorem 5.13 . . . 141
B.6.1. Simulation of AAXML by TAXML . . . 141
B.6.2. Simulation of TAXML by AAXML . . . 141
B.7. Proof of Theorem 5.14 . . . 142
B.8. Proof of Theorem 5.22 . . . 143
B.8.1. Evaluating an FO formula . . . 144
B.8.2. Generating the new artifact tuple . . . 144
B.9. Proof of Theorem 5.23 . . . 145

C. Résumé de la thèse en français 147
C.1. Introduction . . . 147
C.2. Résumé de la partie I : Maintenance de vues sur les documents actifs . . . 148
C.2.1. Résumé du Chapitre 2 : Axlog Model, satisfiabilité et pertinence . . . 149
C.2.2. Résumé du chapitre 3 : Implémentation du modèle d'Axlog . . . 150
C.3. Résumé de la partie II : le modèle AXML Artefact . . . 150
C.3.1. Résumé du chapitre 5 : Comparaison de l'expressivité de langages de spécifications de workflows: un problème de vues . . . 151
C.3.2. Résumé du Chapitre 6: Implémentation du modèle AXML Artefact . . . 154
C.4. Conclusion . . . 154
C.4.1. Gestion de flux : le modèle Axlog . . . 155
Worklow centré données: le modèle AXML Artefact . . . 155
C.4.2. Perspectives . . . 155

List of Algorithms 161

List of Figures **163**

List of Algorithms

2.1. Program $\widehat{P}_q$ for Query q of Figure 2.5 20

A.1. Satisfiability for queries 96
A.2. Scenario for queries 99
A.3. Algorithm Count-RT 105

List of Figures

1.1. An arbitrary join tree-patern query 9

2.1. Updating of an active document 15
2.2. An example of active documents 15
2.3. Examples of queries 16
2.4. A query and some active documents 19
2.5. A query and some active documents 19
2.6. A tree and its reduced tree 23
2.7. Nonmonotonicity: Time and negation 26
2.8. Queries and active documents for relevance 27

3.1. An Axlog widget 33
3.2. The Dell supply chain 33
3.3. Beyond MagicSet 34

5.1. A BAXML document. 51
5.2. Two patterns 52
5.3. Example of a relative query 53
5.4. An instance with an *eval* link 55
5.5. Call guards of *Reject* and *Deliver*. 57
5.6. Example of pattern automaton 58
5.7. Cost of various simulations from Theorems 5.12 and 5.13 60

6.1. The standard workflow of a FRA (solid), its exception (dotted) and its specialization (bold) 68
6.2. Form snapshot for a stage (Audition Scheduled) 70
6.3. Artifacts in the Dell application 71
6.4. An AXML artifact 73
6.5. Architecture of a peer of the AXART System 75

A.1. Examples of queries 96
A.2. An tree coding the triangle graph 118
A.3. A timed query and a timed document 120
A.4. The construction for Theorem 2.18 of φ 123

B.1. Relation adorned with some functions 125
B.2. (i) Pattern whose negation forbids two activated calls and (ii) ensuring satisfaction of $[A_i] \subseteq [A_j]$ 126
B.3. Tree illustrating some of the possible actions for the return of f 132
B.4. The tree some of the actions for the simulation of the activation of the call of f . 136
B.5. The tree of some of the actions for the simulation of the return of the internal call of f 139

B.6. The tree of some of the actions for the simulation of the return of the call of the external function f 140

C.1. A BAXML document. 153