On the Equivalence of Distributed Systems with Queries and Communication

Serge Abiteboul, Balder ten Cate, Yannis Katsis

On the Equivalence of Distributed Systems with Queries and Communication [*]

Serge Abiteboul
INRIA Saclay, ENS Cachan
firstname.lastname@inria.fr

Balder ten Cate [†]
UC Santa Cruz
btencate@ucsc.edu

Yannis Katsis
INRIA Saclay, ENS Cachan
firstname.lastname@inria.fr

ABSTRACT

Distributed data management systems consist of peers that store, exchange and process data in order to collaboratively achieve a common goal, such as evaluate some query. We study the equivalence of such systems. We model a distributed system by a collection of Active XML documents, i.e., trees augmented with function calls for performing tasks such as sending, receiving and querying data. As our model is quite general, the equivalence problem turns out to be undecidable. However, we exhibit several restrictions of the model, for which equivalence can be effectively decided. We also study the computational complexity of the equivalence problem, and present an axiomatization of equivalence, in the form of a set of equivalence-preserving rewrite rules allowing us to optimize a system by rewriting it into an equivalent, but possibly more efficient system.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed databases

General Terms
Algorithms, Languages, Theory

Keywords
Active XML, Equivalence, Optimization

1. INTRODUCTION

Distributed data management has been an important domain of research almost since the early days of databases [12]. With the development of the Web and the emergence of universal standards for data exchange, this problem arguably became a most essential challenge to the database community. We consider systems that store, exchange and apply queries over data, typically to collaborate towards a common goal such as answering a query. In this work, we study the *equivalence* of such systems.

To model these systems we consider an abstraction of the Active XML algebra of [3] that we call *AXML system*. An AXML system (a system for short) is a labeled, unordered, unranked tree, which, apart from extensional data (i.e., regular tree-structured data) also captures *communication* and *query evaluation*. Communication is modelled through send and receive nodes attached to communication channels. Send nodes send data to a channel and all receive nodes attached to that channel receive them. This captures m-to-n point communications and in particular the exchange of data involved in making external function calls. Finally, a system may also include query nodes that capture query evaluation over the data.

The systems we consider are by design very general to capture many use cases. They are distributed, recursive (since the send and receive nodes break the hierarchical structure of the data trees), essentially asynchronous (since send/query operations may happen in arbitrary order while the trees evolve during the computation[1]) and operate on data streams (since queries are activated in a continuous manner producing streams of results). Finally, they may also have external input channels intuitively corresponding to user inputs, data received from other systems, sensors, etc.

Equivalence Results. Unsurprisingly (due to the generality of our model), we can show that equivalence is undecidable in general, even if the queries used inside the system are from quite basic query languages. However, we are able to establish positive decidability results for several restrictions of the model. First, we prove that equivalence is in PTIME in the absence of queries. The limit of a system (i.e., intuitively, the result of exhaustively activating all send and query nodes) is in general infinite, even in the absence of query nodes, but we show that in this context, one can find a finite graph-based representation of a limit and decide system equivalence by comparing these representations.

When queries are introduced, the problem clearly depends on the query language considered. We provide a PTIME decision procedure for testing equivalence of input-free systems containing tree pattern queries (without value-based joins). Joins and inputs further complicate the problem: Joins allow expressing a much richer class of queries. With inputs, on the other hand, the difficulty lies in that for two systems to

[*]This work has been partially funded by the FP7 European Research Council grant agreements Webdam number 226513 and FOX number FP7-ICT-233599.

[†]The author has been partially funded by the NSF grant IIS-0905276

[1]In this paper, we consider only monotone queries. Therefore, the activations of send, receive and query nodes can only enrich the state.

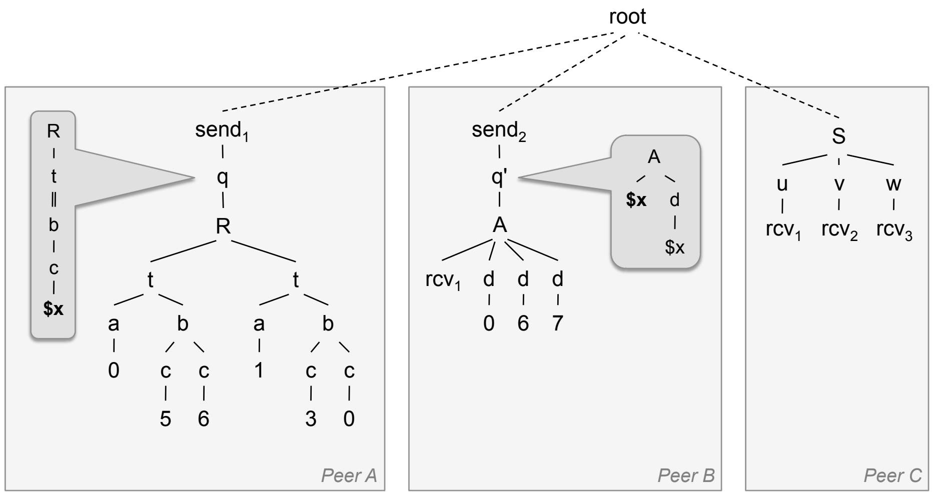

Figure 1: Example of an AXML system

be equivalent in presence of inputs, they must have similar limits for all possible inputs. Nevertheless, we provide decision procedures for testing equivalence of input-free systems containing tree pattern queries with joins, and of systems with input and with tree pattern queries but without joins. The problem remains open when both extensions are considered simultaneously. However, we show decidability for the case of tree pattern queries with XPath-joins (i.e, the tree pattern queries with joins that are expressible in downward XPath with path equalities, cf. the definition of FOXPath in [6]). Finally, the equivalence problem becomes undecidable in the case of tree pattern queries with constructors (even in the absence of joins and input).

Axiomatization. One important application for equivalence checking (and one of the main motivations for this work) is the optimization of distributed systems. Since optimizations have to preserve equivalence, a common approach is to use equivalence-preserving rewrite rules (a.k.a. axioms). [4] presented such axioms for AXML systems. In this work, we go a step further, by presenting a very general set of axioms that furthermore can be shown to be complete for proving equivalence of query-free systems. In other words, given two equivalent query-free systems, it is guaranteed that one can be transformed to the other through a finite number of applications of our axioms.

The paper is organized as follows: In Section 2 we start by defining AXML systems. In Section 3 we present an overview of our results, followed by the actual equivalence results in Sections 4 and 5 (for systems without and with queries, respectively) and our axiomatization in Section 6. Finally in Section 7 we discuss related work and conclude the paper.

2. FRAMEWORK

In this section, we introduce the model studied in the paper, which is an abstraction of the AXML algebra of [3].

2.1 AXML Systems

Active XML systems are finite node-labeled trees, that, apart from regular nodes, may also include (i) *query* nodes to model query evaluation and (ii) *send* and *receive* nodes to capture communication. To distinguish between regular nodes (which describe data extensionally) and send, receive, and query nodes (which describe data intensionally), we refer to the former as *passive* and to the latter as *active* nodes.

More formally, consider the following disjoint alphabets: $\mathcal{L}$ an infinite set of (passive) labels and $\mathcal{C} = \mathcal{C}_{int} \cup \mathcal{C}_{inp}$ an infinite set of channel names, partitioned into *internal channels* and *input channels*. Also consider a monotone query language $\mathcal{Q}$ over XML trees. In this work, we consider queries that, given an XML tree as input, return a set of XML trees. We call a query q *monotone* if whenever an XML tree I homomorphically embed into an XML tree J, then every XML tree in $q(I)$ homomorphically embeds into a XML tree in $q(J)$ (see below for a formal definition of homomorphisms). We will see precise examples of such languages later on. Finally, let $\mathcal{A}$ be the set $\{rcv_c \mid c \in \mathcal{C}\} \cup \{send_c \mid c \in \mathcal{C}_{int}\} \cup Q$ of all allowed active node labels (note that send nodes cannot be indexed by input channels).

Definition 1 (AXML System). *An* AXML system *(a system for short) I is a pair (T, λ), where $T = (N, E)$ is a finite, unordered, unranked tree with nodes N and edges E and $\lambda : N \to \mathcal{L} \cup \mathcal{A}$ is a labeling function over the nodes of the tree, such that (a) only leaf nodes are assigned labels of the form rcv_c and (b) the label of the root is in $\mathcal{L}$.*

In practice, a system may consist of more than one trees. Moreover, these may be distributed over multiple peers. However, for the purposes of this work, it suffices to model the entire system by a single tree (whose root can be seen as a virtual element pointing to the roots of the individual trees). All results can be trivially generalized to systems consisting of multiple trees.

Example 1. *Figure 1 shows an example system. It contains passive nodes, query nodes annotated with tree pattern queries and send/receive nodes grouped into channels (for instance, the two nodes labeled rcv_1 are listening to the channel to which node $send_1$ is submitting). The system has two internal channels (i.e. 1 and 2) and one input channel (channel 3). Although our example includes 3 peers, each with a separate tree, we model the entire system as a single tree with a new virtual root.*

A system without any active nodes is called simply an *XML document* or *XML tree.*

Remark. *In reality, nodes in XML documents and AXML systems have not only a label but also associated atomic data in the form of text and attribute values. To simplify exposition, we do not take the atomic data into account explicitly. Instead, we identify them with node labels. In particular, we work with an infinite set of node labels $\mathcal{L}$ and consider query languages that perform joins on these labels. This choice is not essential, and we could have equivalently worked with other representations, such as data trees (as in [8]).*

Given a system, we are sometimes interested only in its extensional data (i.e. in its static part). The subset of a system containing only the extensional data is called its *snapshot* and it can be derived from the original system by removing all active nodes and their descendants (which intuitively form the arguments of the active nodes). Formally:

DEFINITION 2 (SYSTEM SNAPSHOT). *The* snapshot *of an AXML system I, denoted $I_{\downarrow}$, is the XML document obtained by removing from I all subtrees rooted at active nodes.*

Given two systems we can define homomorphisms between them in the standard way.

DEFINITION 3 (HOMOMORPHISM & ISOMORPHISM). *An AXML system I* maps homomorphically into *a system I', denoted by $I \xrightarrow{hom} I'$, if there exists a homomorphism from I to I', that is, a map from nodes of I to nodes of I' sending the root of I to the root of I' and preserving child-edges and node-labels. Two AXML systems I, I' are* homomorphically equivalent*, denoted by $I \xleftrightarrow{hom} I'$, if $I \xrightarrow{hom} I'$ and $I' \xrightarrow{hom} I$. Finally, I, I' are* isomorphic*, denoted $I \cong I'$, if there is a homomorphism from I to I' that is a bijection.*

In this paper, the monotone queries that we will consider, will essentially see homomorphically equivalent trees as identical, i.e., undistinguishable. So, we can restrict our attention to *reduced* systems. We borrow the following notion of reduced systems from [1]:

DEFINITION 4 (REDUCED SYSTEM). *A system I is said to be* reduced *if there does not exist a homomorphism h from I into itself, such that the range of h is a strict subset of I.*

In graph theory and finite model theory, reduced trees, and more generally, reduced finite structures, are known as *cores* [11]. It is known that for every system I there is a unique (up to isomorphism) reduced system I' to which it is homomorphically equivalent, and that I' is in fact the smallest system that is homomorphically equivalent to I. Moreover, I' can be computed from I in PTIME (by successively removing nodes and checking homomorphism, since the homomorphism problem for trees is known to be in PTIME [9]).

In the rest of this document, whenever we speak of systems, we will assume that they are reduced, unless we explicitly say so otherwise.

2.2 Semantics of AXML Systems

Since a system may contain active nodes, it may evolve over time as these nodes are invoked. In this section, we describe this evolution and use it to define the notion of equivalence between two systems.

The evolution of a system may happen in three ways. First, one can *invoke a send on an internal channel.* A snapshot of the children of this send node is taken and this data is sent over the channel to all the corresponding receive nodes. Secondly, one can *invoke a query node.* The query is evaluated over the snapshot of the children of this query node. The resulting XML trees are appended as siblings of the query node. Finally, one can *invoke the receive nodes of an input channel.* This results in some finite forest of finite XML trees (i.e., an input) being received over this channel.

Observe that only XML documents (i.e., passive trees) are sent over channels. In general in AXML, active trees may also be exchanged (which is useful as it enables *call by name* evaluations strategies, as opposed to *call by value*). For ease of exposition, however, we limit our attention here to the exchange of passive trees only. However, it can be shown that all our results would continue to hold if we were to also allow exchange of active trees.

In this intuitive definition, the snapshots of the children of a node (query or send) play an important role. This motivates the following formal definition: For each system I, the *content* of a node n in I, denoted $content(n, I)$ or $content(n)$ when I is understood, is defined by:

- if n is passive, $content(n)$ is the snapshot of the system rooted in n. (In particular, the snapshot of a system is the content of its root.)
- if n is active, $content(n) = \{content(m) \mid m \text{ is a passive child of } n\}$.

The following definition formalizes one step of the evolution of a system:

DEFINITION 5. *Let $I = (T, \lambda)$ be a system. We say that I can be* transformed to *a system I'* in a single step*, denoted $I \to I'$ iff for some active node n of I, the following happens:*

- *(Send) $\lambda(n) = send_c$ and I' is the (reduced) system derived from I by appending the XML trees in $content(n)$ as siblings of all nodes n' s.t. $\lambda(n') = rcv_c$.*
- *(Query) $\lambda(n) = q \in \mathcal{Q}$ and I' is the (reduced) system derived from I by appending the XML trees in $q(content(n))$ as siblings of n.*
- *(External receive) $\lambda(n) = rcv_c$ for some input channel c and I' is the (reduced) system derived from I by appending the XML trees in some finite forest K as siblings of all nodes n' s.t. $\lambda(n') = rcv_c$. The pair (c, K) of the channel and the input received on it is called the* type *of the external receive action.*

Observe also that in all three cases, the snapshot evolves in a monotone manner, i.e., the system keeps increasing (or staying the same) as a result of the firing of active nodes.

EXAMPLE 2. *Figure 2 illustrates three consecutive transformations of a system $I_0 \to I_0^1 \to I_0^2 \to I_0^3$, showcasing the three possible ways of transforming a system. In each intermediate system, the node in bold indicates the active node whose invocation led to the particular system. Moreover, circles indicate the nodes that have been appended to the system due to this action. In particular, in the first step,*

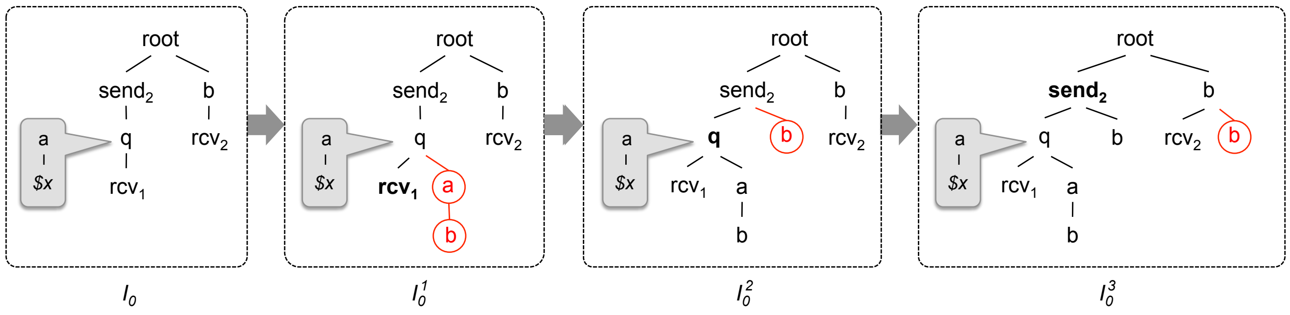

Figure 2: Consecutive transformations of an AXML system I_0

I_0^1 is generated from system I_0 by an external receive from channel 1. In the next step, the received data are queried, leading to system I_0^2. Finally, the result of this query is sent via an internal channel, yielding I_0^3.

A sequence of invocations transforms a system into a different system. We call such a sequence a *run* of the system.

Definition 6 (System Run). *Let I be a system and $\mathcal{I} = (c_{i_1}, F_1)...(c_{i_m}, F_m)$ be a sequence of finite XML forests F_j to be received as inputs on the corresponding input channels c_j occurring in I. Then, I is* transformed for *input $\mathcal{I}$* to *a system I'* in multiple steps, *denoted as $I \xrightarrow{*}_{\mathcal{I}} I'$, iff there exists a sequence of systems $I_1, ..., I_n$ ($n \geq 1$) such that $I = I_1 \rightarrow ... \rightarrow I_n = I'$ and $\mathcal{I}$ is the sequence of the types of the external receive actions occcurring in this sequence of transformations. Such a sequence is referred to as a* run *of the system I.*

When the input sequence is understood, we simply write $I \xrightarrow{*} I'$. Clearly, there may be several runs starting from the same instance I even for the same inputs. Furthermore, even without inputs and without queries, an instance may evolve in different manners and it may have arbitrarily long runs. However, because of the monotonicity of the queries, all fair infinite runs converge to the same infinite tree. The fairness condition involved here is that any action that can be fired, eventually is. This is illustrated next in an example.

Example 3. *Consider the system I_1 of Figure 3a. It is easy to see that this system allows for infinite runs, since the node send$_1$ supplies the content of send$_2$ with data, which in turn it receives back in an augmented form, due to the presence of rcv$_2$ in the subtree of send$_1$. This allows creating progressively deeper trees by iteratively activating send$_1$ and send$_2$. One can see that all such fair infinite runs converge to the tree depicted in Figure 3b:*

$J_1 : \{root\{b\{c, a\{b\{c, a\{b\{...\}\}\}\}\}\}\}$.

However, some unfair infinite runs may instead converge to

$J_2 : \{root\{b\{a\{b\{a\{b\{...\}\}\}\}\}\}\}$.

by never activating send$_3$.

It will become clear what we mean exactly by convergence in Section 4, where we will give the formal definition of a limit of a system.

Having defined a run of a system, we can now define the equivalence of two systems. Intuitively, two systems I and J are equivalent if on any arbitrary input, whenever I can be transformed to I', J can be transformed to a system J' that subsumes I' and vice versa. Formally:

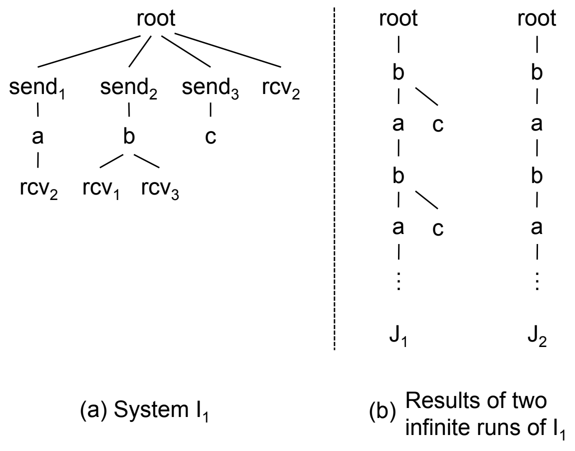

Figure 3: Possible results of runs

Definition 7 (Equivalence). *Let I, J be two AXML systems. Then I, J are* equivalent *if for each finite input sequence $\mathcal{I}$, (i) for every run $I \xrightarrow{*}_{\mathcal{I}} I'$ there is a run $J \xrightarrow{*}_{\mathcal{I}} J'$ such that $I'_{\downarrow} \xrightarrow{hom} J'_{\downarrow}$, and (ii) for every run $J \xrightarrow{*}_{\mathcal{I}} J'$ there is a run $I \xrightarrow{*}_{\mathcal{I}} I'$ such that $J'_{\downarrow} \xrightarrow{hom} I'_{\downarrow}$.*

The following proposition provides evidence that this is indeed a natural notion of equivalence. We say that a Boolean monotone query q is *eventually true* in a system I for some input $\mathcal{I}$ if there is a run $I \xrightarrow{*}_{\mathcal{I}} I'$ such that $q(I'_{\downarrow}) =$ true. The following result shows that two equivalent systems are essentially indistinguishable by monotone queries:

Proposition 1. *Two AXML systems, I, J, are equivalent if and only if for each finite input sequence $\mathcal{I}$ and for each monotone query q, q is eventually true in I for input $\mathcal{I}$ if and only if q is eventually true in J for input $\mathcal{I}$.*

Note that q above is universally quantified over all possible monotone queries and not only those occurring in I or J.

3. OVERVIEW OF RESULTS

The main focus of the present paper is the study of the *Equivalence Problem* for AXML systems, i.e., the problem of testing whether two systems are equivalent. In Sections 4 and 5, we study this problem for different classes of AXML systems. Each such class is identified by choices along two

	No input	Input
No queries	PTIME	PTIME
Tree Pattern Queries (TPQs)	PTIME	3EXPTIME; PSPACE-hard
TPQs with XPath-joins	PTIME	3EXPTIME; PSPACE-hard
TPQs with Arbitrary Joins	$P_{\|\|}^{NP}$-complete	open
TPQs with Constructors	undecidable	undecidable

Figure 4: Complexity results for equivalence

orthogonal axes: Firstly, the query language Q that is considered, and secondly, the presence or absence of input channels. Figure 4 summarizes our results on the complexity of the Equivalence Problem, with the vertical axis for the choice of query language, and the horizontal one for the consideration of input channels.

The results highlight that the introduction of input channels complicates the equivalence problem. This is not surprising since to prove equivalence, we have to verify that the two systems have similar limits for all possible inputs. It also shows that (as usual) joins greatly increase the power of the query language and, in our case, the complexity of the equivalence problem. The presence of constructors also complicates the problem. Intuitively, we can use such constructors to "create space" for computations.

Finally, in Section 6 we present a complete axiomatization for query-free AXML systems, in the form of finitely many equivalence-preserving rewrite rules that can be used to transform a system into any other equivalent system.

4. QUERY-FREE SYSTEMS

In this section we consider the equivalence of query-free systems. This is nontrivial, because equivalence is defined in terms of runs (see Definition 7) and (i) a system may have infinitely many runs even with no input and (ii) a single run may be itself infinite. We therefore need to tackle this infinite dimension. In the rest of this section we look first at input-free systems and then to those with inputs.

Query-free & input-free systems. Given two query-free and input-free systems to compare, we will show that it suffices to consider their limits. The limit of a system, intuitively, is the possibly infinite XML document towards which the successive snapshots on a run of the system converge, assuming that the run is fair. We will then see that these limits, even when infinite, can be represented in a finite manner and exhibit an algorithm working on these finite representations that decides equivalence.

In fact, our definition of limits will not explicitly speak of fair runs, because this would require us to give a definition of fair runs, which is slightly tedious (however, the reader may verify that our definition of limits can indeed be equivalently cast in terms of fair runs). We say that an infinite tree I^* is the limit of a system I if the snapshot of each instance reachable from I can be embedded in I^* and conversely, each finite prefix of I^* can be embedded in the snapshot of some instance reachable from I. More formally,

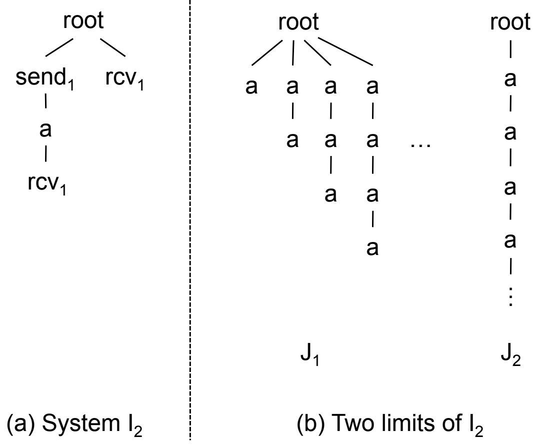

Figure 5: Limits of an AXML system

Definition 8 (Limit of an AXML system). *Let I be an AXML input-free system. We say that an infinite tree I^* is a* limit *of I, if (i) whenever $I \xrightarrow{*} I'$, then $I'_{\downarrow} \xrightarrow{hom} I^*$ and (ii) for every finite height prefix I^*_k of I^*, there is an I' with $I \xrightarrow{*} I'$, such that $I^*_k \xrightarrow{hom} I'_{\downarrow}$.*

A system might in general have more than one limit.

Example 4. *For instance, consider the system I_2 shown in Figure 5a. It is easy to see that the infinite trees J_1, J_2 in Figure 5b are two limits of this system. Observe that J_2 is finitely branching, whereas J_1 is not.*

As illustrated by the example, infinitely branching limits can be counterintuitive: if the limit of a system is to represent the document towards which an the snapshots of systems in an infinite fair run converge, we would not expect to get an infinitely branching tree, since each transformation of a system involves reducing it. In what follows, we will only consider finitely branching limits. As we will see, this happens without loss of generality, since every system has a finitely branching limit.

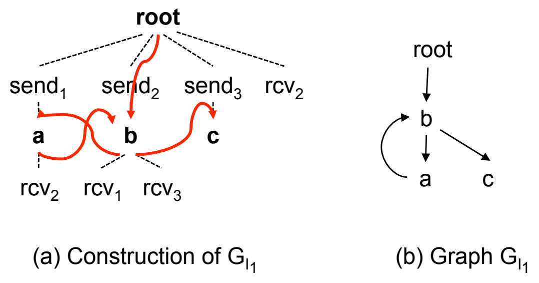

Figure 6: Graph G_{I_1} for system I_1 of Figure 3

It turns out that our definition of equivalence (which involves a possibly infinite number of runs), can be equivalently cast in terms of the limits of two systems, as shown next. Observe the important subtlety, namely the "finite branching" requirement.

Proposition 2. *Let I, J be input-free AXML systems having finitely branching limits I^* and J^*. Then I and J are equivalent if and only if $I^* \xleftrightarrow{hom} J^*$.*

Proof. Suppose that I and J are equivalent. It follows from the definition of limits that, then, every finite height prefix of I^* must have a homomorphism into J^* (and vice versa). It is a general fact of finitely branching trees that if this is the case, then there is also homomorphism from the entire tree I^* to J^* (and vice versa). In other words, $I^* \xleftrightarrow{hom} J^*$. Conversely, suppose that $I^* \xleftrightarrow{hom} J^*$. Let us denote by I^*_k and J^*_k the prefix of I and of J of height k. Then, in particular, (since homomorphism must send roots to roots and preserve the child relation) $I^*_k \xleftrightarrow{hom} J^*_k$ for all k. Combining this with the fact that I^* is a limit for I and J^* a limit for J, we obtain that, whenever $I \xrightarrow{*} I'$, there is a J' with $J \xrightarrow{*} J'$ such that $I'_\downarrow \xrightarrow{hom} J'_\downarrow$. Similarly, in the other direction. In other words, I and J are equivalent. □

This proposition highlights the interest of finitely branching limits of systems. However, we will not reason directly on these possibly infinite limits. Instead, we exhibit here a finite representation of such limits. For this, we use finite, labelled, directed, rooted graphs. In the following, we simply call them "graphs". Formally, a *graph* is a tuple (N, E, r, λ), where N a finite set of nodes, $E \subseteq N \times N$ the set of edges, λ a labeling function over N, and $r \in N$ the root.

A graph G represents a possibly infinite tree, that we call the *unraveling* of G and denote by $unr(G)$, as follows: The nodes of $unr(G)$ are all non-empty finite sequences $x_1 \ldots x_n$ where each x_i is a node of G, x_1 is the root, and for each $i < n$, there is an edge from x_i to x_{i+1} in G. Furthermore, if $x_1 \ldots x_n$ is a node of $unr(G)$, then it is a child of the node $x_1 \ldots x_{n-1}$ (that is also clearly a node of $unr(G)$) and the label of $x_1 \ldots x_n$ in the tree is that of x_n in G.

We next associate to each input-free, query-free system I, the graph G_I graph obtained by (i) adding an edge from the parent of each receive node to each child of every send node attached to the same channel and then (ii) dropping all send and receive nodes together with all other nodes that are not reachable from the root anymore.

Example 5. *For instance, Figure 6b shows the graph G_{I_1} obtained for system I_1 of Figure 3. Figure 6a illustrates its construction. The dotted edges are those that have been dropped. The thicker arcs are those that have been added.*

The following results states that this graph represents a limit of the system.

Proposition 3. *For every input-free, query-free AXML system I, $\text{unr}(G_I)$ is a finitely branching limit of I.*

Note, that the limit represented by $unr(G_I)$ is a finitely branching one, allowing us to utilize Proposition 2.

To conclude the analysis of equivalence for such systems it therefore suffices to show how to decide homomorphic equivalence between the unravelling of two graphs. For this, we use the auxiliary notion of graph simulation. A graph G *simulates* a graph H if there is a binary relation Z between nodes of G and nodes of H, called a *simulation*, satisfying the following conditions: (i) $(root_G, root_H) \in Z$, (ii) whenever $(x, y) \in Z$, then x and y have the same label, and (iii) whenever $(x, y) \in Z$ and y has a successor y' in H, then x has a successor x' in G such that $(x', y') \in Z$. We say that two graphs are *simulation-equivalent* if they simulate each other. Now we have:

Proposition 4. *For all graphs G, H, $\text{unr}(G) \xrightarrow{hom} \text{unr}(H)$ if and only if H simulates G, which can be tested in* PTIME.

Thus to decide equivalence between two such systems, we can first build their corresponding graphs in linear time and then check homomorphic equivalence between their unravelings using simulation on the graphs (which is also in PTIME). This leads to:

Theorem 1. *Equivalence for AXML query-free, input-free systems is in* PTIME.

For future reference, we mention one further fact about graphs and simulations: We call a graph G *minimized* if (i) every node is reachable from the root; (ii) there are no two distinct children n, m of a node in the graph, such that n simulates m (i.e., such that G with n taken as the root simulates G with m taken as the root); and (iii) no two distinct nodes are simulation-equivalent.

Proposition 5. *Every graph is simulation-equivalent to a minimized graph; two minimized graphs G, G' are simulation-equivalent if and only if they are isomorphic.*

We will put this to use when we consider axiomatization.

Query-free systems with input. As already mentioned, the introduction of input channels complicates the equivalence problem in general. Query-free systems are the exception. Because of the absence of queries, the system cannot "look inside" the data provided by an input channel. Such data ends up behaving as a single "black block of data" that may end up replicated as-is in possibly many places. Therefore, we can treat input channels simply as fresh symbols, reducing thus the equivalence problem in the presence of inputs to that in their absence. More formally, we have:

Theorem 2. *For any AXML query-free system I (with input channels), let $\widehat{I}$ be the AXML system obtained from I by uniformly replacing each receive node $recv_i$ from some input channel i by a fresh (passive) label $\blacksquare_i$. Then two systems I, J are equivalent if and only if $\widehat{I}$ and $\widehat{J}$ are equivalent. Thus, equivalence of query-free AXML systems can be tested in* PTIME.

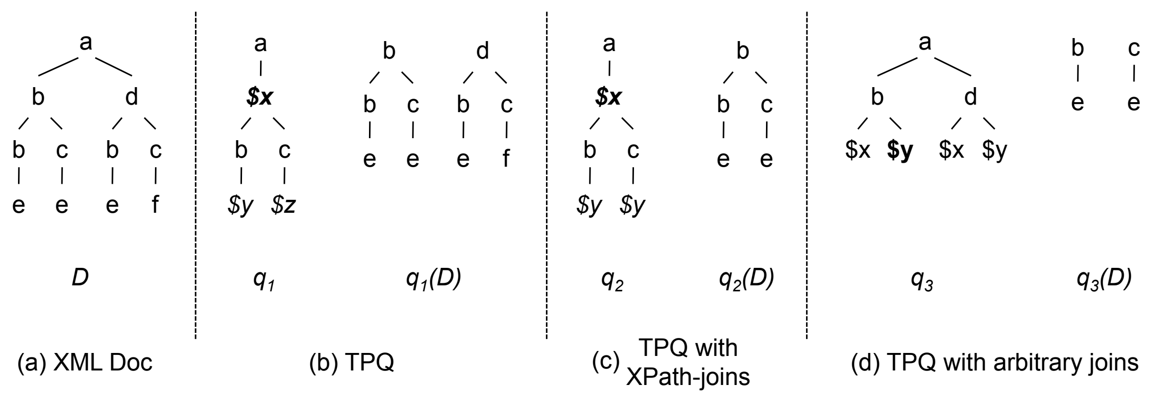

Figure 7: Examples of TPQs with joins

Another way to put it is as follows: In testing equivalence of query-free systems, active node labels rcv_i (where i is an input channel) may be ignored and simply treated as (distinct) passive labels.

5. AXML SYSTEMS WITH QUERIES

The equivalence problem becomes harder when we allow systems that contain query nodes. In this section we consider such systems. Since the complexity of checking equivalence depends on the query language employed, we study different query languages. We start by defining these languages and then present the equivalence results; first for input-free systems and then for systems with input.

5.1 Query languages

The query languages we consider are all variants of tree pattern queries [5].

TPQs with Joins. We first provide the definition of tree pattern queries with joins (TPQ-J).

Definition 9 (Tree Pattern Queries with Joins). *Let $\mathcal{V}$ be an infinite set of variables. A* tree pattern query with joins (TPQ-J) *is a tree whose edges are labeled by* child *or* descendant*, and whose nodes are labeled by elements of $\mathcal{L} \cup \mathcal{V}$, together with a distinguished "result node" corresponding to the root of the subtree to be returned by the query.*

The semantics is the following: Let q be a TPQ-J and I an XML document. A *matching* of q in I is a map sending nodes of q to nodes of I and variables from $\mathcal{V}$ to labels from $\mathcal{L}$, such that (i) the root of q is mapped to the root of I, (ii) child/descendant relationships and labels from $\mathcal{L}$ are preserved, (iii) for each node of q labeled by a variable from $\mathcal{V}$, the image of the label of the node is the label of the image of the node. Evaluating q on I yields the set of all subtrees J of I for which there is a matching from q to I such that the result node of q is mapped to the root of J.

Clearly, whenever a TPQ-J q has more than one occurrence of the same variable, it is performing a join. We say that q is a *tree pattern query without joins* or simply a *tree pattern query* (TPQ) if it does not contain two occurrences of the same variable. Note that, in this case, the only role of a variable is to act as a wildcard.

A *TPQ with XPath-joins* (TPQ-XJ) is a TPQ-J satisfying the following structural condition (†):

(†) Call a node x in a tree pattern an *intermediary* of a pair of nodes y, z if y and z are joining nodes (i.e., are labeled by the same variable), x lies on the shortest path between y and z (which includes y and z themselves), and x is not the least common ancestor of y and z. The following two conditions hold:

1. no node is an intermediary of two different pairs of nodes,
2. no node on the path from the root to the result node is an intermediary of any pair of nodes.

Although its definition is arguably somewhat involved, the condition (†) is quite natural. Indeed, it can be seen that the TPQs with XPath-joins (i.e., the TPQs with joins satisfying the above condition) are precisely the TPQs with joins that can be expressed in downward navigational XPath with path equalities (as defined in [8]).[2] TPQs with XPath-joins are well-behaved in other respects as well. For instance, seen as conjunctive queries, they have a bounded tree-width [6].

Example 6. *Figure 7 shows examples of tree pattern queries and their results on an example XML document. Labels of the form $x (shown in italics) represent variables and labels in bold face signify a pattern's result node. Figure 7b shows a TPQ without joins and Figures 7c and 7d show TPQs with joins. While the query in Figure 7c is a TPQ with XPath-joins, the query in Figure 7d is not, as it fails the first condition.*

TPQs with Constructors. Apart from tree pattern queries with joins, we also consider tree pattern queries with constructors. Instead of being allowed to simply copy a single subtree appearing in its input to its output, a tree pattern query with constructors can create and output a new tree constructed from existing data.

Definition 10 (TPQs with Constructors). *A tree pattern query with constructors (TPQ-C) is a pair (q, t)*

[2]This is an observation we made together with Diego Figueira (personal communication).

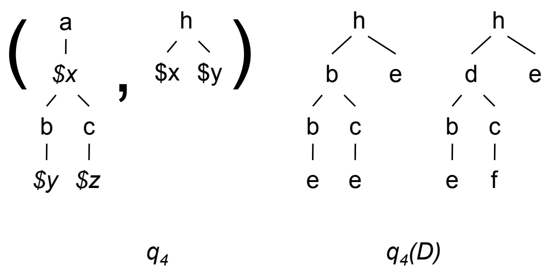

Figure 8: Example of TPQ with constructors

where q is a TPQ and t a template, i.e., an XML document in which the labels of some of the leaves have been replaced by variables occurring in q.

Example 7. *Figure 8 shows an example of such a query and its result when applied on document D of Figure 7a.*

The semantics is defined as follows: Let I be an XML document and let $q' = (q, t)$ be a TPQ-C. Each matching m of q in I, yields an answer t_m obtained by replacing in t each leaf labeled $x \in \mathcal{V}$ by the subtree of I rooted at $m(s)$, where s is the node of q labeled x. Finally, $q'(I)$ is the set of all t_m, where m is a matching of q in I.

Note that the result node of q is irrelevant in this definition. Also, note that the present definition is only meaningful for TPQs without joins because it uses the fact that each variable occurs only once in the query. The definition could be adapted to also incorporate joins also, but as it will turn out that TPQ-Cs quickly lead to undecidability even without joins, the above definition is sufficient.

5.2 Input-free Systems

For input-free systems with queries, a graph representing a limit of the system cannot be obtained directly from the system as before, because, intuitively, computing the limit involves repeatedly evaluating the queries of the system.

However, it turns out that for systems with TPQs with joins, we can construct a datalog program that computes such a graph. Moreover, if the joins are limited to XPath-joins, then the datalog program can be executed in polynomial time. Therefore, we can test equivalence in polynomial time by executing the datalog programs and testing whether the graphs obtained simulate each other.

On the other hand, as we will show later in this section, for input-free systems containing TPQs with constructors, equivalence is undecidable.

TPQs with Joins. We look first at the case of TPQs with joins. As a first step, we specify the datalog program computing the graph that represents a limit of the system. In what follows, let I be an input-free system containing TPQs with joins. Let G_I be the graph associated to I (treating query nodes as ordinary nodes). We now define Π_I to be the following datalog program, where m, n, and a stand for constants, whereas x, y, z are variables.

child(m,n) :-	(for E_G(m,n) & $\lambda(n) \in \mathcal{L}$)
label(m,a) :-	(for $\lambda(m) = a \in \mathcal{L}$)
child(m,y) :- child(n,x), q(x,y)	(for E_G(m,n) & $\lambda(n) = q$)
desc(x,y) :- child(x,y)	
desc(x,y) :- child(x,z), desc(z,y)	

Here, by $q(x, y)$ we denote the query q seen as a conjunctive query over EBDs "child", "desc", and "label", where x is identified with the root, and y with the result node.

We denote by $\widehat{G}_I$ the graph obtained by running the datalog program Π_I until it reaches a fixed point. In this way, we get an analogue of Proposition 3 for AXML systems with TPQs with joins:

Proposition 6. *For every input-free AXML system with TPQs with joins, $unr(\widehat{G}_I)$ is a finitely branching limit of I.*

The proof of Proposition 6 crucially uses the fact that the queries we consider are bisimulation invariant, and hence evaluating the queries on the unraveling of a graph yields the same result as evaluating the queries on the graph itself and unraveling the result.

Given the result of the previous section, this immediately yields an algorithm for testing equivalence of two systems. Since the algorithm involves constructing and running a datalog program, it requires in general more than polynomial time. However, it can be shown that it runs in time $\mathrm{P}^{\mathrm{NP}}_{||}$, i.e., deterministic polynomial time with parallel access to an NP-oracle. This follows from the fact that the arity of the IDBs in the constructed datalog program is bounded by a constant (in fact, the constant in question is 2), and it is not difficult to show that the combined complexity of evaluating a datalog program whose IDBs have bounded arity is in $\mathrm{P}^{\mathrm{NP}}_{||}$. In fact, we have the following:

Theorem 3. *The equivalence problem for input-free AXML systems with TPQs with joins is $\mathrm{P}^{\mathrm{NP}}_{||}$-complete.*

In particular, this shows that testing equivalence of AXML systems with TPQs with joins is NP-hard. Indeed, $\mathrm{P}^{\mathrm{NP}}_{||}$ contains the entire Boolean hierarchy, of which NP is only the first level.

This holds for arbitrary joins. If we restrict our attention to TPQs with XPath-joins, then we can prove much better complexity bounds. These are based on the result shown in [6] that TPQs with XPath-joins, seen as conjunctive queries, have *bounded tree-width*. It follows that, in this case, the constructed datalog program Π_I has bounded tree-width, in the sense that the body of each rule has bounded tree-width. Without going into the definition of tree-width, what matters here is that the complexity of testing for a homomorphism between two structures is known to be in PTIME if the first structure has bounded tree-width [9] (cf. [7] for more details). It follows, that evaluating a datalog program of bounded tree-width whose IDBs have bounded arity, is PTIME. Consequently, the above algorithm for testing equivalence of two systems runs in PTIME if the systems only contain TPQs with XPath-joins.

Theorem 4. *The equivalence problem for input-free systems with TPQ with XPath-joins is decidable in* PTIME.

TPQs with Constructors. Finally, we move to systems containing TPQs with constructors. In this case, equivalence turns out to be undecidable, as these systems are expressive enough to simulate the computation of a Turing Machine. In particular, we show the undecidability by reduction from the acceptance problem of a Turing Machine (TM): Given a TM and an input, we create an AXML system that simulates the TM on the input and returns a designated symbol if and only if the TM accepts this input. Hence we obtain the following undecidability result:

Theorem 5. *Equivalence of input-free AXML systems with TPQs with constructors is undecidable.*

In fact, the undecidability proof uses no input channels, no joins, and no repeated variables in the template.

Remark. *We could also consider an alternative semantics of the language with constructors, where we would replace the leaves of the template by the matching* node *instead of the entire subtree. One can still show undecidability of equivalence for this query language (by reduction from the containment of datalog) if inputs and joins are allowed.*

5.3 Simplifying Systems with Input

We consider next systems with input. In this case, our strategy of creating a graph representing a limit of the system is no longer directly applicable. The reason is that queries can now operate on inputs, which are unknown beforehand. Therefore we can only hope to create a finite representation of a limit if we allow it to contain also queries over the input. Indeed, we will employ graphs that contain queries and our decision procedure for equivalence will reason on them.

To simplify the equivalence check, we will first bring the systems into a special form. For our purposes, a *simple* system will be one in which all queries have been pushed down to the inputs. The procedure that we use to transform a system into an equivalent simple system is interesting in its own right, and we first describe it in its own dedicated section. Then in the following subsection, we will present the decision procedure for equivalence for two simple systems.

Regular Tree Pattern Queries. As we will see, to be able to push queries down to the inputs we need to employ a query language that is more powerful than TPQs. To this end we generalize the query languages we use, by allowing a limited form of recursion.

Definition 11. *A regular tree pattern query (RTPQ) is a regular expression over the infinite alphabet consisting of all tree pattern queries. RTPQs with Joins and RTPQs with XPath-joins are defined in the same way.*

Intuitively, we need this additional expressive power to capture the recursion that would otherwise be modelled by the send and receive nodes, if the queries were allowed to appear at arbitrary places in the tree (and not only directly over the inputs).

Example 8. *Consider the $send_1$ subtree in Figure 9a. It is easy to see that it sends to channel 1 all subtrees of the input channel* ext *that are reachable by an $(a/b)^*$-path. Let I be the system consisting of a root whose children are this same $send_1$ subtree, and a rcv_1 node. The system I is not simple, as the query q_1 is applied over an internal channel. However, it is equivalent to the system $\{root\{q\{rcv_c\}\}\}$, where q is the RTPQ $/(a/b)^*$. This illustrates the fact that, in order to turn a system with TPQs into an equivalent simple system, it may be necessary to introduce RTPQs.*

Simple systems. A system is simple if it only applies queries directly on the input, and not on pieces of XML that have been produced by the system itself. It turns out that a system can apply queries on the input in two ways: It can either copy part of the input to the output or simply check whether the input contains a pattern to output something else (i.e. perform a boolean check on the input). These two ways are demonstrated by the following example:

Example 9. *Consider the two query nodes of system I_3 in Figure 9a. As we discussed above, q_1 simply copies part of the input channel ext to the output. On the other hand, q_2 uses the input differently. It simply checks whether ext contains the symbol γ and if it does, it returns c followed by the contents of the internal channel 1.*

In order to give a formal definition of simple systems, we introduce two new types of active nodes, that represent these two types of queries: $\textit{input-query}_{c,q}$ and $\textit{input-test}_{c,q}$, where c is an input channel and q is a query. The first, which is only allowed to occur as a leaf, is shorthand for $q\{rcv_c\}$. In other words, it corresponds to copying part of the input to the output. The second is allowed to occur anywhere and $\textit{input-test}_{c,q}\{data\}$ is shorthand for $q'\{s\{in\{rcv_c\}, out\{data\}\}\}$, where q' is the query $/s[in/q]/out$, which tests whether q has a non-empty result in the subtree below the *in*-node, and then returns the data below the *out*-node. In other words, $\textit{input-test}_{c,q}\{data\}$ returns a copy of *data* if query q yields a non-empty result on the data received on input channel c, and nothing otherwise.

We call an AXML system *simple* if it only uses queries in these two ways, i.e., if it can be seen as an AXML system using the active node labels $send_i$, rcv_i as usual, and, instead of arbitrary query nodes, the active node labels $\textit{input-query}_{c,q}$ and $\textit{input-test}_{c,q}$.

Example 10. *Figures 9b and 9c show a simple system equivalent to I_3, using the original notation and the newly introduced active nodes, respectively.*

Making systems simple. To simplify the statement of the next result, we will restrict our attention to systems with a single input channel. This is harmless, because if a system has input channels $c_1, \ldots, c_n$, one can replace each rcv_{c_i} by $q_i(rcv_c)$, where c is a single input channel and q is the query $/i$. Although this changes the semantics of the system, two systems are equivalent before this modification if and only if they are equivalent after. Therefore, in studying the complexity of the equivalence problem we may assume a single input channel.

The following theorem shows that every system is equivalent to a simple system, provided that the queries are allowed to use recursion and the system has a single input channel.

Theorem 6. *Given a system with TPQs that has a single input channel, one can compute an equivalent simple system with RTPQs. Similarly for systems with TPQs with XPath-joins. In both cases the translation can be carried out in 2EXPTIME.*

The proof involves a detour through monadic datalog [10]. More specifically, we identify a class of monadic datalog

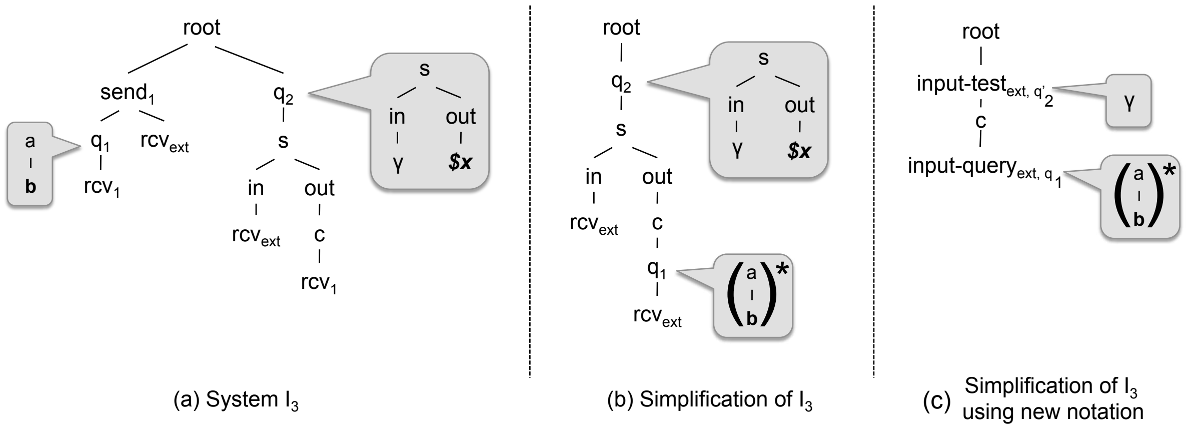

Figure 9: Example of system simplification

queries that can express precisely the RTPQs (with XPath-joins). We then prove the Theorem using these monadic datalog queries.

RTPQs are clearly more powerful than TPQs as a query language. Indeed, RTPQs enhance the expressive power of simple systems, as compared to TPQs. Interestingly, this is not the case for arbitrary (non-simple) systems with inputs, as shown by the following result:

Theorem 7. *Every AXML system with RTPQs can be translated in polynomial time to an equivalent AXML system with TPQs. Similarly for RTPQs with joins, and for RTPQs with XPath-joins.*

Rather than giving a formal proof we illustrate Theorem 7 by means of an example. Consider the regular tree pattern $(q_a q_b)^*$, where q_a is the tree pattern that selects a-children of the root, and q_b is the tree pattern that selects b-children of the root. In other words, $(q_a q_b)^*$ selects all nodes reachable by an $(ab)^*$-path from the root. In order to construct an AXML system computing this query, we first translate the regular expression to a non-deterministic finite state automaton (NFA). In this case, the NFA A has two states, 1 and 2, and a transition from 1 to 2 labeled by the tree pattern q_a, and a transition from 2 to 1 labeled by the tree pattern q_b. State 1 is both the initial state and the final state. Now, from the NFA A we construct an AXML system I_A. It has one channel for each state of the automaton, plus the external input channel. In this case, the system is $\{send_1\{q_a\{rcv_2\}, rcv_c\}, send_2\{q_b\{rcv_1\}\}, rcv_1\}$.

Theorem 7 shows that the recursion natively supported by all systems through the interaction of send and receive nodes, is a very powerful construct. It allows, among others, systems with TPQs to express regular path languages by simulating finite state automata.

5.4 Simple Systems with Input

It follows from results in [8] that the containment problem for unions of RTPQs with XPath-joins is decidable. In fact, a slightly more general result holds:

Proposition 7 ([8]). *The following is decidable in EXPTIME: Given two unions of RTPQs with XPath-joins q, q', and a Boolean combination ϕ of Boolean RTPQs with XPath-joins, does $q \subseteq q'$ hold on XML documents satisfying ϕ?*

Here, a *Boolean RTPQ with XPath-joins* is a RTPQ with XPath joins viewed as a Boolean (non-emptiness) query. Note that unions of RTPQs with XPath-joins, as well as Boolean combinations of such Boolean queries, can be directly expressed in $RegXPath(\downarrow, =)$; the main logic studied in [8]. We use Proposition 7 to show that the equivalence problem for simple systems with input and RTPQs with XPath-joins is decidable (and therefore also the equivalence for *non-simple* such systems).

Theorem 8. *The equivalence problem for simple AXML systems with input and RTPQs with XPath-joins is decidable in EXPTIME.*

Proof sketch. The basic outline of the proof is as follows. Let I, J be the systems that we need to test for equivalence. We collect all Boolean queries occurring in I and J in the form of *input-tests*. We consider, one by one, all combinations of these Boolean queries, and, for each case, test if I and J are equivalent on all inputs satisfying exactly those Boolean queries (if so, then, indeed, I and J must be equivalent on all possible inputs). This allows us to eliminate *input-tests* from I and J, so that we only have to test (many) equivalences between simple systems that do not contain any *input-test* nodes. Finally, in order to test whether two simple systems with *input-queries* are equivalent (on a restricted class of inputs) we construct graphs representing the limits of the two systems, where the nodes of the graphs may be annotated by queries over the input, and we test simulation-equivalence of the graphs (but taking into account containment relations that may hold between queries). Below, we spell out this approach in more detail.

Suppose I, J are simple systems with input and TPQs with XPath-joins. Let Q_{test} be the set of queries occurring in I and J as *input-tests*, and let Q_{query} be the set of queries occurring in I and J as *input-queries*.

For each subset $X \subseteq Q_{test}$, let I^X and J^X be obtained by removing all *input-test* nodes, and adding edges from the parent of an *input-test* node to all its children if the *input-test* query is in X. Note that I^X and J^X are simple systems without *input-test* joins that behave in exactly the same way as I and J do, on inputs for which it holds that X is exactly the set of queries from Q_{test} that are satisfied. Hence, in order to test whether I and J are equivalent on all inputs, it is enough to test that each I^X is equivalent to J^X on inputs satisfying exactly those queries from Q_{test} that belong to X.

This leaves us with the task of showing that the following problem is decidable in EXPTIME (note that performing exponentially many EXPTIME-tasks is still in EXPTIME).

> Given simple systems I, J without *input-tests*, and given a Boolean combination ϕ of Boolean RTPQs with XPath-joins, decide if I and J are equivalent on inputs satisfying ϕ.

We say that a subset $Y \subseteq Q_{query}$ is closed w.r.t. ϕ if the following holds: For all $q \in Q_{query}$, if $q \subseteq \bigcup_{q' \in Y} q'$ on XML documents satisfying ϕ, then $q \in Y$. Suppose we are interested in the behavior of a system I on input satisfying ϕ. Then, we may assume that for every node of I, the set of all *input-query* children of that node form a closed set of queries that is a closed subset of Q_{query} with respect to ϕ (if not, then the relevant additional *input-query* children can be added without affecting the semantics of I on inputs satisfying ϕ). In this case, we say that I is closed w.r.t. ϕ. Proposition 7 allows us to compute the closure of I w.r.t. ϕ in EXPTIME (in the size of I and ϕ) by repeatedly testing containment and until no further *input-query* children need to be added to the document.

Finally, let I, J be simple documents without *input-tests*, ϕ be a Boolean combination of Boolean RTPQs with XPath-joins, and I, J be closed with respect to ϕ. Then it can be seen that I and J are equivalent with respect to ϕ if and only if the graphs of I and J are simulation-equivalent, where the *input-query* nodes are now treated as passive nodes (each query is treated as a different symbol). Indeed, if the graphs of I and J are not simulation equivalent, then I and J have different (non-homomorphically equivalent) limits on any input XML data satisfying ϕ that is *distinguishing* in the sense that it contains data satisfying any closed combination of queries from Q_{query}.

Since the existence of simulations can be tested in PTIME, we get an overall upper bound of EXPTIME. □

Combining this with Theorem 6, yields the following result for deciding equivalence of (non-simple) systems with input and TPQs with XPath-joins:

COROLLARY 1. *The equivalence problem for (non-simple) AXML systems with input and TPQs with XPath-joins is decidable in 3EXPTIME.*

We do not know whether the 3EXPTIME-bound is tight. However, we know that the equivalence problem for systems with TPQs and inputs is PSPACE-hard.

THEOREM 9. *The equivalence problem for simple systems with RTPQs and input, and hence also the equivalence problem for systems with TPQs and input, is PSPACE-hard.*

This follows directly from the PSPACE-hardness of the equivalence problem for regular expressions.

Remark. *In the case where the joins are not restricted to XPath-joins, decidability of equivalence still remains open. However, for such systems equivalence is decidable if we restrict our attention to inputs over a fixed set of labels. The result is based on the fact that then a join can be replaced by a disjunction of finitely many patterns in which the join variable is replaced by a concrete value (and therefore the problem is reduced to the join-free case).*

6. AXIOMATIZATION FOR QUERY-FREE AXML SYSTEMS (WITH INPUTS)

As a first step in studying optimization of AXML systems, we present here a finite set of axioms (or, more precisely, a finite set of axiom schemes) that can be used to rewrite systems into other, equivalent systems. More importantly, we show that these axioms are complete for query-free AXML systems, in the sense that for every two such systems I, J, if I and J are equivalent, then I can be rewritten into J by a finite sequence of applications of the axioms as undirected rewrite rules. The axioms are the following:

ax1	$send_c\{data1, data2\} = send_c\{data1\}, send_c\{data2\}$
ax2	$send_c\{data1\}, a\{data2\} = a\{send_c\{data1\}, data2\}$
ax3	$send_c\{data\}, rcv_c = send_c\{data\}, data$ if there is no other $send_c$
ax4	$send_c\{data\} = \epsilon$ if there is no rcv_c
ax5	$send_c\{rcv_{c'}, data\} = send_c\{data\}$ if every rcv_c node in the document has a $rcv_{c'}$ sibling
ax6	$send_c\{data\} = \epsilon$ if c is an inaccessible channel
ax7	$rcv_c rcv_d = rcv_c$ if channel c simulates channel d
ax8	$rcv_c = rcv_d$ if channels c and d simulate each other

The axioms ax6-ax8 require some explanation, as they refer to "inaccessible channels" and to simulations between channels. However, before we get to that, we state a result concerning the axioms ax1-ax5.

In what follows, whenever we speak of systems, we always assume that they are query-free. We call a system *normalized* if it is a tree where all subtrees immediately below the root are of the form $send_c\{a\{rcv_{c1}, \ldots, rcv_{cn}\}\}$ where a is a single node, and such that there do not exist two send nodes for the same channel. A normalized system can naturally be seen as an encoding of a graph, where the channels are the nodes of the graph and each subtree of the form $send_c\{a\{rcv_{c1}, \ldots, rcv_{cn}\}\}$ specifies the incoming edges of the node corresponding to channel c. Indeed, the graph represented by a normalized AXML system I, in this way, is precisely the graph of I as we defined it in Section 4.

As a first step, we have:

LEMMA 1. *Using the axioms ax1-ax5 as undirected rewrite rules, every system can be rewritten to a normalized system.*

PROOF HINT. The axiom ax1 is used to make sure there is a single send node per channel and the axiom ax2 is used for moving around send nodes and bringing them directly below the root of the system. The axioms ax3 and ax4 are used for splitting up data into pieces consisting of a single label, and introducing intermediate channels. Finally, ax5 (in combination with ax2 and ax3) is used to ensure guardedness, i.e. that every *rcv*-node is below a passive node and not below a *send*-node. □

We say that a channel c is *accessible* in a system, if there is a sequence of channels $c_1, \ldots, c_n$ such that $c_n = c$, c_1 occurs in the system in a place that is not in the scope of any *send*-node, and each c_{i+1} occurs in the scope of some $send_i$-node. Intuitively, a channel is accessible if data sent on this channel will eventually affect the shapshot of the system. A *simulation* in a normalized system is a binary relation Z between channels such that whenever $Z(c, d)$ and the system contains $send_d\{a\{rcv_{d_1}, \ldots, rcv_{d_n}\}\}$ then there are channels $c_1, \ldots, c_n$ such that the system contains $send_c\{a\{c_1, \ldots, c_n\}\}$ and Zc_id_i holds for all $i \leq n$. If there is a simulation Z such that Zcd, then we say that c simulates d. This explains the notation in axiom ax6-ax8.

We say that a normalized document is *minimized* if (i) every channel is accessible, (ii) no two different channels simulate each other, and (iii) it does not contain siblings rcv_c and rcv_d where c simulates d. It is clear that we have the following lemma:

LEMMA 2. *Every normalized AXML system can be rewritten to a minimized normalized AXML system using the axioms ax6-ax8 as undirected rewrite rules.*

Now, it follows from Proposition 5 that two minimized normalized AXML systems are equivalent if and only if they are isomorphic. Hence, we have

THEOREM 10. *Two query-free AXML systems are equivalent if and only if one can be rewritten to the other using the axioms ax1-ax8 as undirected rewrite rules.*

PROOF. The right-to-left direction corresponds to the fact that the axioms are sound (which can be easily seen). For the left-to-right direction, suppose that I, J are equivalent AXML systems. Let I', J' be minimized normalized AXML systems such that I, I' are provably equivalent and J, J' are provably equivalent. Then I' and J' coincide (recall that we identify AXML systems up to isomorphism). $\square$

In the case of acyclic systems (systems where the dependencies between the channels do not induce a cycle), fewer axioms are needed. In this case, the axioms ax2-ax4 allow us to effectively eliminate all *send*-nodes and *rcv*-nodes for internal channels. This means we have to consider documents with external receive nodes only. It follows by Theorem 2 that any two such systems are equivalent if and only if they are homomorphicaly equivalent, i.e., they are isomorphic when reduced. Since we identify systems up to homomorphic equivalence, this gives us

THEOREM 11. *Two acyclic AXML systems are equivalent if and only if one can be rewritten to the other using the axioms ax2-ax4 as undirected rewrite rules.*

7. DISCUSSION

We conclude by summarizing our main results, putting them into perspective, and discussing related work.

The main motivation of this work was providing formal foundations for the optimization of distributed systems with queries and communication (which we model as AXML systems). To this end, we identified a well-behaved notion of equivalence and investigated the complexity of testing equivalence for different classes of AXML systems. Our complexity results, ranging from PTIME to undecidability, show that we cover a large spectrum of AXML systems in terms of expressive power.

Our framework is based on the work on Positive AXML [1], which identified monotone AXML systems as a well-behaved class of AXML systems. Our results rely implicitly on properties shown in [1], such as confluence (which implies that all infinite fair runs yield the same system).

In addition to providing decision procedures for equivalence, we also studied the axiomatization of AXML systems. In particular, we presented a complete set of axioms for the equivalence of query-free systems. Although there is more work to be done in this direction (generalizing the result to systems with queries is an interesting problem for future work), this is an important first step in addressing formally the optimization problem for AXML systems, and it is a natural continuation of the work on OptimAX [4], which presented a (sound but not complete) set of rewrite rules for AXML systems.

At this point, we would like to mention that some of the results we obtained along the way are of independent interest, either because they may serve as a stepping stone in further analysis of AXML systems, or because they provide further insight into the capabilities and limitations of AXML systems. In particular, our results in Section 5.3 show that it is possible to push queries appearing in an AXML system down to the input. We believe that this is an important step towards understanding issues such as relevance (i.e. which parts of the input are relevant to the result of an AXML document) [2]. The same results in Section 5.3 also characterize in some sense the expressive power of AXML systems. They show, for example that the queries that are computable by AXML systems containing tree pattern queries, are precisely the *regular tree-pattern queries*. Regular tree-patterns extend tree-patterns with a limited form of recursion, and allow us to express queries such as "return all nodes reachable by an $(ab)^*$-path from the root".

Acknowlegements. We are grateful to Pierre Bourhis and Diego Figueira for useful discussions.

8. REFERENCES

[1] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *PODS*, pages 35–45, 2004.

[2] S. Abiteboul, P. Bourhis, and B. Marinoiu. Satisfiability and relevance for queries over active documents. In *PODS*, pages 87–96, 2009.

[3] S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *EDBT*, pages 1049–1058, 2006.

[4] S. Abiteboul, I. Manolescu, and S. Zoupanos. OptimAX: Optimizing Distributed ActiveXML Applications. In *ICWE*, pages 299–310, 2008.

[5] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD*, pages 497–508, 2001.

[6] M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1):1–54, 2008.

[7] V. Dalmau, P. G. Kolaitis, and M. Y. Vardi. Constraint Satisfaction, Bounded Treewidth, and Finite-Variable Logics. In *CP*, pages 310–326, 2002.

[8] D. Figueira. Satisfiability of downward XPath with data equality tests. In *PODS*, 2009.

[9] E. C. Freuder. Complexity of K-Tree Structured Constraint Satisfaction Problems. In *AAAI*, pages 4–9, 1990.

[10] G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for Web information extraction. *J. ACM*, 51(1):74–113, 2004.

[11] P. Hell and J. Nešetřil. The Core of a Graph. *Discr. Math.*, 109:117–126, 1992.

[12] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 2nd edition, January 1999.