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

Serge Abiteboul[a], Balder ten Cate[b], Yannis Katsis[a]

[a] *INRIA Saclay, ENS Cachan*
[b] *UC Santa Cruz*

Abstract

Distributed data management systems consist of peers that store, exchange and process data in order to collaboratively achieve a common goal, such as evaluating 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.

1. Introduction

Distributed data management has been an important domain of research almost since the early days of databases [14]. 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. A major question in such systems is optimization: How can one rewrite a system into another equivalent system (i.e., a system that computes the same result) that is more efficient? In order to answer this question, we have first to study *equivalence*: When are two systems equivalent and how can we decide equivalence? This is

[☆]This work has been partially funded by the FP7 European Research Council grant agreements Webdam number 226513 and FOX number FP7-ICT-233599. The second and third authors have also been partially supported by the NSF grants IIS-0905276 and IIS-1117527, respectively. We thank Diego Figueira for his help with Proposition 5.

Email addresses: firstname.lastname@inria.fr (Serge Abiteboul), btencate@ucsc.edu (Balder ten Cate), firstname.lastname@inria.fr (Yannis Katsis)

the topic of this paper. In this work, we define the equivalence of distributed systems and present equivalence decidability results for different classes of systems. Moreover, as a first step towards optimization, we also present a complete set of equivalence-preserving rewrite rules (a.k.a. axioms), that allows us to optimize AXML systems and to prove equivalence for a limited class 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) may also contain active nodes capturing *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 this data. This captures m-to-n point communications and in particular the exchange of data involved in making function calls. We distinguish between two types of channels: internal channels and external input (or simply input) channels. Internal channels model communication happening within the system (in which case it is known what data is sent into the channel). On the other hand, input channels model communication arriving from inputs external to the system (in which case the data sent into the channel has to be treated as a black box). Finally, a system may also include query nodes that capture query evaluation over the data. Following [1], we consider *positive* AXML systems, in which all queries are monotone. These have been identified as an important and well-behaved special case in previous literature.

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) and operate on data streams (since queries are activated in a continuous manner producing streams of results). Finally, they may also receive external data, such as user inputs, data from other systems, sensors, etc. (which are modelled through external input channels).

Equivalence Decidability 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 depends on the query language considered, as well as on the presence of external input channels. We provide a PTIME decision procedure for testing equivalence of input-free systems (i.e., systems without input channels) containing tree pattern queries (without value-based joins). Joins and input channels further complicate the problem: Joins allow expressing a much richer class of queries. With input channels, on the

other hand, the difficulty lies in that for two systems with input channels to be equivalent, 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 [7]). Finally, the equivalence problem becomes undecidable in the case of tree pattern queries with constructors (even in the absence of joins and input).

Axiomatization. As explained above, an important application for equivalence testing (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). In [4], several such axioms were presented for AXML systems. 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.

This article is the full version of the conference article [5]. The most significant difference between the two is that the current article provides the proofs missing from [5].

Organization. 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.

2. Framework

In this section, we introduce the model studied in the article, 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*. Finally, let $\mathcal{Q}$ be a query language for

XML trees. We will soon make precise what we mean by a query language, but, for the moment, one may think of $\mathcal{Q}$ as an infinite set of abstract expressions. We denote by $\mathcal{A}$ the set $\{rcv_c \mid c \in \mathcal{C}\} \cup \{send_c \mid c \in \mathcal{C}_{int}\} \cup \mathcal{Q}$. The elements of $\mathcal{A}$ are the "active" node labels that may appear in an Active XML system. Note that $\mathcal{A}$ does not contain a label of the form $send_c$ with c being an input channel. The reason is that, in contrast to internal channels, data sent to input channels is not created by the system itself but instead given to it as input by external sources.

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 $\mathcal{L}$.*

A system without any active nodes (i.e., a system in which all node labels come from $\mathcal{L}$) is called an *XML document* or *XML tree*. Note that we do not consider any sibling order in this paper.

The query languages $\mathcal{Q}$ that we consider in this paper will consist of queries that, given a set of XML trees as input, return a set of XML trees. Furthermore, all queries q will be *monotone*, meaning that, whenever I homomorphically embeds into J (where I and J are XML trees, or, more generally, sets of XML trees) then every XML tree in $q(I)$ homomorphically embeds into a XML tree in $q(J)$ (see below for a formal definition of homomorphisms). The same assumption of monotonicity was made in [1][1]. A typical example of a monotone query language for XML trees is the language of *tree pattern queries* [6]. The monotonicity of tree pattern queries follows directly from the fact that they are conjunctive queries, and the latter are well known to be preserved by homomorphisms [8].

In practice, a system may consist of more than one tree. 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 distributed over multiple peers.

Example 1. *Figure 1 shows an example system. This example is meant to simply demonstrate the structure of an AXML system. Its semantics as well as the query language used will be formally defined in subsequent subsections. The displayed system 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). Channels 1 is an internal channel as there exist labels of the form*

[1] In [1], monotonicity is not defined in terms of homomorphisms, but in terms of set-theoretic containment. However, this distinction is irrelevant given that, in [1] as well as here, attention is restricted to *reduced* trees, as defined below.

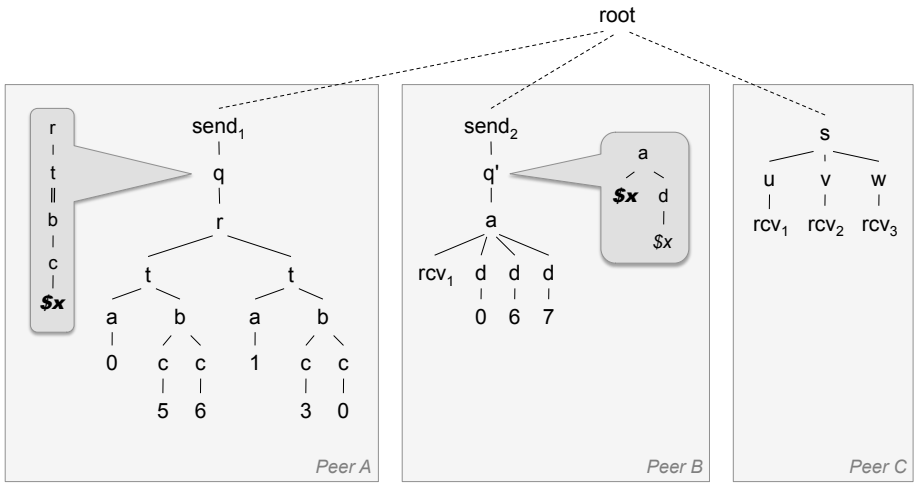


Figure 1: Example of an AXML system

both $send_1$ and rcv_1. Similarly for channel 2. On the other hand channel 3 is an input channel as there is only a node label of the form rcv_3 (and not one of the form $send_3$). Finally, although our example includes 3 peers, each with a separate tree, we model the entire system, as explained above, as a single tree with a virtual root.

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 [10]).*

Given a system, we are sometimes interested only in its extensional data (i.e. in its passive 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 (cf. [1], where homomorphisms are called *subsumptions*).

Definition 3 (Homomorphism and 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 article, the considered monotone queries 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* [13]. 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, due to the fact that I is a tree, 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 [11]).

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 node attached to 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, where it is appended as siblings of the receive node. 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* evaluation 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

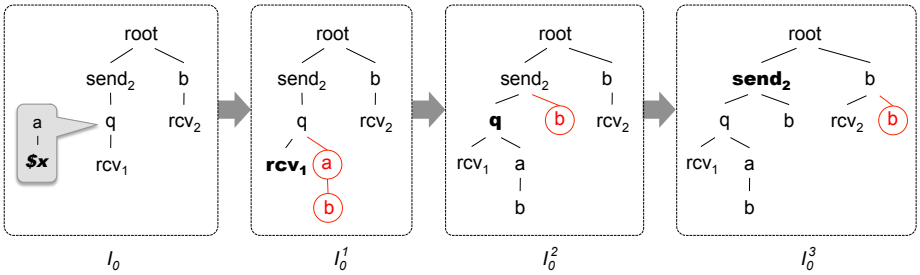


Figure 2: Consecutive transformations of an AXML system I_0

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, one of the following holds:*

- *(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, in the sense that the old system always homomorphically maps into the new system obtained 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 I_0^i, 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, I_0^1

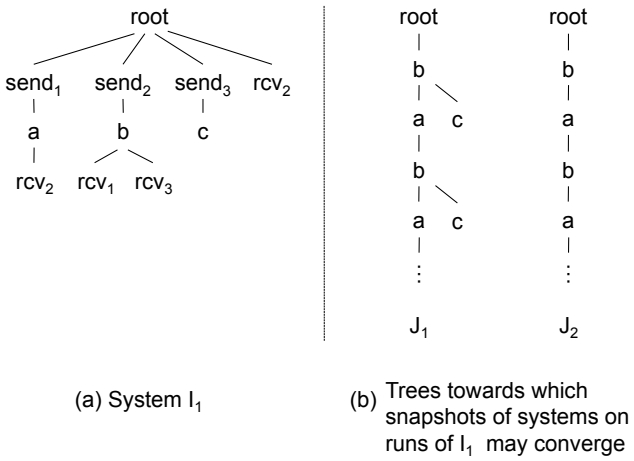


Figure 3: Trees towards which the snapshots of systems reachable by runs may converge

Before presenting the example, let us formally introduce the notation that we will employ to represent trees: A tree t rooted at a is denoted as $a\{t_1, \ldots, t_n\}$, where $t_i, i = 1, \ldots, n$ are the trees rooted at children of a. If a has no children, then t is represented simply as a.

Example 3. *Consider the system I_1 of Figure 3a. It is easy to see that this system allows for arbitrarily long 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 runs lead to systems whose snapshots "converge" to the first tree depicted in Figure 3b:*

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

An unfair run may instead lead to systems whose snapshots converge to the second tree in Figure 3b:

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

by never activating $send_3$.

While following the example, recall that each transformation step of a system leads to a reduced system. This is the reason why for example neither J_1 nor J_2 contain multiple a nodes as siblings. Moreover, note that we are interested in the tree towards which the snapshots *of a system converge and not the system itself (i.e., we ignore the active nodes of a system when we look at its "limit").*

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, equivalence is defined as follows (recall that we denote by $I_\downarrow$ the *snapshot* of an AXML system I):

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 $J'_\downarrow \xrightarrow{hom} I'_\downarrow$.*

Note that the definition of equivalence of two systems is based on the existence of homomorphisms between the *snapshots* of the systems and not between the actual systems. This allows two systems that use completely different communication channels and queries to still be equivalent if their passive part (as defined by their snapshots) intuitively converges to the same "limit".

3. Overview of Results

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

Figure 4: Complexity results for equivalence

The main focus of the present article 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 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. We consider first *input-free* query-free systems (i.e., query-free systems without external input channels), and then query-free systems with input. Note that, even for input-free, query-free systems, the equivalence problem is non-trivial. This is because equivalence is defined in terms of runs (see Definition 7) and a system may have infinitely many, and arbitrarily long, runs.

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 is a possibly infinite tree towards which the successive snapshots on a run of the system, in a precise sense, converge. We will show that these limits, even when infinite, can be represented in a finite manner, and we will present an algorithm that operates on these finite representations in order to decide equivalence.

To be able to talk about the possibly infinite limit of a system, we first need to introduce the notion of an infinite XML tree. An infinite XML tree is an XML tree as defined in Section 2 but with the difference that it contains an infinite number of nodes. Definition 3 of homomorphism and isomorphism can be straightforwardly extended to infinite XML trees.

We say that an infinite XML tree I^* is a limit of a system I if the snapshot of each instance reachable from I can be embedded in I^* and conversely, each finite height prefix of I^* can be embedded in the snapshot of some instance reachable from I. Here, by a *finite height prefix* $t|_k$ of a tree t (where k is a natural number) we mean the tree containing only those nodes from t having distance at most k from the root.

Definition 8 (Limit of an AXML system). *Let I be an AXML input-free system.*[2] *We say that an infinite XML 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 $I^*|_k \xrightarrow{hom} I'_{\downarrow}$.*

Note that Definition 8 does not tell us how limits are constructed. All it specifies is that a limit of a system I has a certain relationship to the set of all snapshots of systems obtainable from I by means of a run. Moreover, it is important to note that the limit is not an arbitrary AXML system but a (possibly infinite) *XML tree*; i.e., it does not contain active nodes. However, as we will see, it contains enough information to decide equivalence, since equivalence of two systems, as discussed above, is a relationship between the snapshots of the systems (which are XML trees) and not between the systems themselves.

[2] This definition, in principle, applies to input-free systems with or without queries. However, in this section, we will consider only the query-free case.

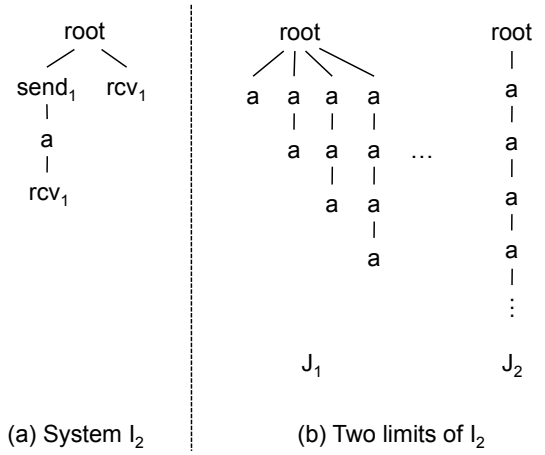


Figure 5: Limits of an AXML system

Example 4. *Consider the system I_2 shown in Figure 5a. It is easy to see that the infinite trees J_1 and J_2 in Figure 5b are each a limit of this system.*

The above example shows that a system may have several homomorphically non-equivalent limits (note that, while $J_1 \xrightarrow{\text{hom}} J_2$, it is *not* the case of $J_2 \xrightarrow{\text{hom}} J_1$, i.e., J_1 and J_2 are not homomorphically equivalent). We call a possibly infinite XML tree *finitely branching* if every node has finitely many children. In what follows, we will only consider finitely branching limits. As we will show, every input-free system has precisely one finitely branching limit, up to homomorphic equivalence. Note that, in the above example, J_2 is finitely branching, whereas J_1 is not.

Furthermore, we will show that the equivalence of two input-free systems (which was defined in terms of a possibly infinite number of runs), can be equivalently cast just in terms of the finitely branching limits of two systems, cf. Proposition 1 below.

We first need to establish a technical lemma.

Lemma 1. *Let I, I' be finitely branching but possibly infinite XML trees. If for every finite height prefix $I|_k$ of I, $I|_k \xrightarrow{hom} I'$, then $I \xrightarrow{hom} I'$.*

Proof. Let $h_k : I|_k \xrightarrow{\text{hom}} I'$, for $k \geq 1$. We will construct the desired homomorphism $g : I \xrightarrow{\text{hom}} I'$ as an infinite union $\bigcup_k g_k$ of partial homomorphisms, where $g_k : I|_k \xrightarrow{\text{hom}} I'$, such that (i) $g_k \subseteq g_{k+1}$ for all $k \geq 1$, and (ii) for each $k \geq 1$ there are infinitely many $\ell > k$ for which it holds that $g_k \subseteq h_\ell$. The first condition ensures that the union $\bigcup_k g_k$ is well-defined, while the second condition is a convenient invariant used in the construction of the homomorphisms $g_1, g_2, \ldots$. Specifically, let g_1 be the map that sends the root node

of I to the root node of I' (which indeed satisfies both conditions). Suppose that we have constructed $g_1 : I|_1 \xrightarrow{\text{hom}} I', \ldots, g_k : I|_k \xrightarrow{\text{hom}} I'$ satisfying the above conditions. There are only finitely many possible homomorphisms $g_{k+1} : I|_{k+1} \xrightarrow{\text{hom}} I'$ (here, we use the fact that I and I' are finitely branching) and hence, by the pigeon hole principle, there is at least one such homomorphism $g_{k+1} : I|_{k+1} \xrightarrow{\text{hom}} I'$ extending g_k that is contained in infinitely many h_ℓ with $\ell > k$. We can pick any such. Continuing this way, we obtain the desired sequence of homomorphisms $g_1 \subseteq g_2 \subseteq \ldots$ of which the union $\bigcup_k g_k$ is a homomorphism from I to I'. □

Proposition 1. *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 $I^* \xleftrightarrow{hom} J^*$. In particular, every input-free AXML system has at most one finitely branching limit, up to homomorphic equivalence.*

Proof. Suppose that I and J are equivalent. It follows from the definition of limits that, then, every finite height prefix $I^*|_k$ of I^* must have a homomorphism into J^*, i.e., $I^*|_k \xrightarrow{\text{hom}} J^*$ (and vice versa). Hence, by Lemma 1, there is also a homomorphism from the entire tree I^* to J^* (and vice versa). In other words, $I^* \xleftrightarrow{\text{hom}} J^*$. Conversely, suppose that $I^* \xleftrightarrow{\text{hom}} J^*$. Then, in particular, (since a homomorphism must send roots to roots and preserve the child relation) $I^*|_k \xleftrightarrow{\text{hom}} J^*|_k$ 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 $I'_\downarrow \xrightarrow{\text{hom}} J'_\downarrow$. Similarly, in the other direction. In other words, I and J are equivalent. □

This proposition highlights the relevance of finitely branching limits of systems. However, we still have to establish the *existence* of finitely branching limits. This is what we will show next (cf. Proposition 3 below). In fact, we will exhibit a finite representation of these, possibly infinite, limits. For this, we use finite, labelled, directed, rooted graphs. In the following, we simply call them "graphs". Formally, a *(rooted) 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. The tree-structure of $unr(G)$ is defined as follows: x_1 (which is a sequence of length 1) is the root of $unr(G)$; if $x_1 \ldots, x_n$ is a node of $unr(G)$ with $n > 1$, then the (unique) *parent* of this node is the sequence $x_1 \ldots x_{n-1}$ which is indeed also a node of $unr(G)$; and the label of $x_1 \ldots, x_n$ is the label of x_n in G.

For every node $x_1 \ldots x_n$ of $unr(G)$, by its *original* we will mean the node x_n of G. So, in particular, the label of a node of $unr(G)$, by the above definition, is the label of its original in G.

Given two graphs, it is possible to decide if the infinite trees they represent (i.e., their unravellings) are homomorphically equivalent. 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 $(x', y') \in Z$. Note that Z is not required to be a function, not every node of G is required to belong to the domain of Z, and not every node of H is required to belong to the co-domain of Z (although every node of H that is reachable from the root will belong to the co-domain of Z). We say that two graphs are *simulation-equivalent* if they simulate each other. Now we have:

Proposition 2. *For all graphs G and H, we have that* $\mathrm{unr}(H) \xrightarrow{hom} \mathrm{unr}(G)$ *if and only if G simulates H, which can be tested in* PTIME.

Proof. Let $G = (N_G, E_G, r_G, \lambda_G)$ and $H = (N_H, E_H, r_H, \lambda_H)$.

For the left-to-right direction, for each node $n \in N_G$, we will use G_n to denote the graph derived from G by considering n as the root, i.e., $G_n = (N_G, E_G, n, \lambda_G)$. Similarly, we define H_n for $n \in N_H$. To establish the left-to-right direction, it is enough to observe that the binary relation $\{(n, n') \mid unr(H_{n'}) \xrightarrow{\mathrm{hom}} unr(G_n)\}$ is itself a simulation, which is easy to verify.

For the right-to-left direction, let Z be a simulation of G by H. We define a homomorphism $h : unr(H) \xrightarrow{\mathrm{hom}} unr(G)$. Recall that the nodes of $unr(G)$ are sequences σ of nodes of G starting with r_G. We define $h(\sigma')$ by induction on the length of the sequence σ', and in such a way that, whenever $h(\sigma') = \sigma$, with $\sigma' = x'_1 \dots x'_n$ and $\sigma = x_1 \dots x_n$, then $(x_n, x'_n) \in Z$. First, $h(r_H) = r_G$. Next, suppose $h(x'_1 \dots x'_n) = (x_1 \dots x_n)$ and suppose $x'_1 \dots x'_n x'_{n+1}$ is an element of the domain of $unr(G)$. Then x'_{n+1} is a successor of x'_n in H, and hence, by the definition of simulations, there is a node x_{n+1} of H such that x_{n+1} is a successor of x_n in G and $(x_{n+1}, x'_{n+1}) \in Z$. We pick an arbitrary such node x_{n+1} and extend h by sending $x'_1 \dots x'_n x'_{n+1}$ to $x_1 \dots x_n x_{n+1}$. It is clear that the function h obtained in this way is a homomorphism.

Existence of a simulation can be tested in PTIME using a greatest-fixed point computation: Start with the total binary relation, and keep removing pairs that falsify one of the conditions until either the result is a simulation, or it no longer contains the root-pair (in which case there is no simulation). □

We next associate to each input-free, query-free system I a graph G_I, whose unravelling, we will show, is a finitely branching limit of I. Given an AXML system I, by a *chain of communication channels* (a *chain* for short) we mean a sequence $c_1, \dots, c_n$ of channels ($n \geq 1$), such that for all $1 \leq i < n$, there is a send node attached to channel c_i with a child that is a receive node attached to channel c_{i+1}. The intuition is that if there is a chain $c_1, \dots, c_n$, then data sent on channel c_n will eventually be received on channel c_1. The graph G_I is obtained from I as follows: (i) all send and receive nodes are removed, (ii) an

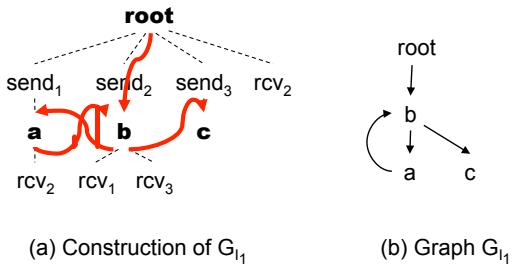


Figure 6: Graph G_{I_1} for system I_1 of Figure 3

edge is added from a (passive) node x to a (passive) node y if there is a chain $c_1, \ldots, c_n$, such that x is the parent (in I) of a receive node attached to channel c_1, and y is a child (in I) of a send node attached to channel c_n, and finally, (iii) all nodes of G_I that are not reachable from the root are removed from the graph.

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. Finally, the nodes in bold face are the ones that have been kept, while the nodes in normal face are the ones that have been dropped.*

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

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

Proof. First of all, $unr(G_I)$ is finitely branching by construction. To show that it is a limit of I, we have to prove the following: (i) Whenever $I \xrightarrow{*} I'$, then $I'_{\downarrow} \xrightarrow{\text{hom}} unr(G_I)$ and (ii) for every finite height prefix $unr(G_I)|_k$ of $unr(G_I)$, there is an I' with $I \xrightarrow{*} I'$, such that $unr(G_I)|_k \xrightarrow{\text{hom}} I'_{\downarrow}$.

Proof of (i). We show that, whenever $I \to I'$, then G_I and $G_{I'}$ are simulation-equivalent. It then follows, by Proposition 2, that $unr(G_I) \xleftrightarrow{\text{hom}} unr(G_{I'})$. This implies the desired result (by transitivity of the homomorphism relation), since $I_{\downarrow} \xrightarrow{\text{hom}} unr(G_I)$. Let $I \to I'$. Since the system I is input-free and query-free, I' is obtained from I by invoking a send node. Clearly, G_I is included in $G_{I'}$ and hence $G_{I'}$ simulates G_I. Conversely, note that if a node x of $G_{I'}$ does not belong to G_I, then x was added as a result of the send node invocation, which means that x is a copy of a node x' in G_I. It can be seen that the binary relation Z containing all identity pairs (x, x) (where x is a node of G_I) as well as all pairs (x, x') where x' is the copy of x created by the send node invocation, is a simulation of $G_{I'}$ by G_I.

Proof of (ii). The proof proceeds by induction on k. It will be convenient to use a slightly stronger induction hypothesis. We first introduce some convenient notation. We use the notation $T|_k$ to denote the height k prefix of a tree T, and we use the notation T_x to denote the subtree of a tree T rooted at a node x. Finally, we denote by $\pi : unr(G) \to G$ the natural homomorphism which sends each node $x_1, \ldots, x_n$ of $unr(G)$ to the node x_n of G. The induction hypothesis that we use is the following:

($*$) For each k, there is an AXML system I' with $I \to^* I'$ such that for each node n of $unr(G_I)$, we have $unr(G_I)_n|_k \xrightarrow{\text{hom}} (I'_{\downarrow})_{\pi(n)}$.

Observe that, if we pick n to be the root, then this induction hypothesis implies the condition (ii) that we are trying to prove.

It is clear that ($*$) holds when $k = 1$. Suppose that the claim holds for some value of k, i.e., there is an instance I' with $I \to^* I'$ such that for each node n of $unr(G_I)$, we have $unr(G_I)_n|_k \xrightarrow{\text{hom}} (I'_{\downarrow})_{\pi(n)}$. Let I'' be the AXML system obtained from I' by invoking all send nodes, one after the other, and repeating this m times, where m is the number of channels occurring in I. Thus each send node is invoked m times. This construction guarantees that, for every chain of communication channels $c_1, \ldots, c_m$, all passive XML data occurring in I' below a send node attached to channel c_m will occur in I'' below every receive node for channel c_1.

We claim that, for any node n' of $unr(G_I)$, $unr(G_I)_{n'}|_{k+1} \xrightarrow{\text{hom}} (I''_{\downarrow})_{\pi(n')}$. The homomorphism in question sends n' to $\pi(n')$. For any child n'' of n' in $unr(G_I)$, we can distinguish two cases: either there is an edge from $\pi(n')$ to $\pi(n'')$ in I, or this edge was added in the construction of G_I, in which case n' and n'' were connected by a chain of communication channels $c_!, \ldots, c_m$ in I. In the first case, we can immediately apply the induction hypothesis, which gives us a homomorphism from $unr(G_I)_{n''}|_k$ to $(I'_{\downarrow})_{\pi(n'')}$. In the second case, we use the induction hypothesis together with the fact that all passive data occurring in I' below a $send_{c_m}$ node occurs in I'' below every rcv_{c_1} node. Finally, we combine all the homomorphisms (for the different children of n') into one homomorphism from $unr(G_I)_{n'}|_{k+1}$ to $(I''_{\downarrow})_{\pi(n')}$. □

Note, that the limit represented by $unr(G_I)$ is a finitely branching one, allowing us to utilize Proposition 1.

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 query-free, input-free AXML systems is in* PTIME.

PROOF. From Propositions 1 and 3, it immediately follows that two input-free, query-free systems I and J are equivalent iff $unr(G_I) \xleftrightarrow{\text{hom}} unr(G_J)$. Since G_I and G_J can be constructed in polynomial time and the homomorphic equivalence can be checked in polynomial time (see Proposition 2), we get the result. □

For future reference (specifically, to prove the completeness of our proposed set of axioms in Section 6), we will prove one more result about graphs and simulations. Given a graph G with nodes n, m, let us say that n *simulates* m if G_n simulates G_m, where G_n is the graph derived from G by letting n be the root, and similarly for G_m. 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; and (iii) no two distinct nodes are simulation-equivalent.

Proposition 4. *Every graph is simulation-equivalent to a minimized graph; two minimized graphs G, G' are simulation-equivalent iff they are isomorphic.*

Proof. First, every graph G is clearly simulation-equivalent to the subgraph containing only the nodes that are reachable from the root.

Next, if two nodes n and m simulate each other, then $unr(G, n) \xleftrightarrow{\text{hom}} unr(G, m)$, where by (G, n) we denote the graph G where n is taken as the root. It follows that $unr(G) \xleftrightarrow{\text{hom}} unr(G')$, where G' is obtained from G by identifying the nodes n and m (i.e., replacing the two by a single node, and connecting all edges that were connected to n or m to the new node). Hence, also G and G' are simulation equivalent.

Finally, suppose that there are two distinct children n, m of a node x, such that n simulates m. Then $unr(G, m) \xrightarrow{\text{hom}} unr(G, n)$. It easily follows that, if G' is the graph obtained from G by removing the edge (x, m), then $unr(G') \xleftrightarrow{\text{hom}} unr(G)$. Therefore G and G' are simulation equivalent.

As for the second claim, if G and G' are isomorphic, they are clearly simulation-equivalent. In the other direction, let Z, Z' be simulations in both directions. The desired isomorphism can be constructed step-by-step. We sketch the construction. Initially we take the partial isomorphism sending the root r of G to the root r' of G'. Next, we consider the children of r on one hand and the children of r' on the other. For each child x of r, let $x^{\rightarrow}$ be a child of r' such that $(x, x^{\rightarrow}) \in Z$. Similarly, for each child y of r', let $y^{\leftarrow}$ be a child of r such that $(y, y^{\leftarrow}) \in Z'$. It follows from the minimality of G and G' (more specifically, condition (ii) of minimality, in combination with the transitivity of the simulation relationship) that $(x^{\rightarrow})^{\leftarrow} = x$ and $(y^{\leftarrow})^{\rightarrow} = y$. Hence, the function $(\cdot)^{\rightarrow}$ defines a bijection between the successors of r and the successors of r' where each pair in the bijection is simulation-equivalent. Repeating this, we finally obtain a bijection, containing only pairs that are simulation-equivalent. But this must be an isomorphism. □

Proposition 4 can be viewed as providing an alternative method for testing the equivalence of two query-free, input-free systems (but less practical, since it would require solving the graph isomorphism problem). However, will put this to use in Section 6 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

though 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 rcv_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.

PROOF. One direction is clear: If I and I' are equivalent then they are in particular equivalent in the case where the input on channel i consists of a single node labeled $\blacksquare_i$, hence $\widehat{I}$ and $\widehat{J}$ are equivalent too.

The other direction follows from the fact that a limit of I on an input $\{(c_1, F_1), \ldots, (c_n, F_n)\}$ is the finitely branching tree obtained by taking the limit of $\widehat{I}$ and replacing each $\blacksquare_i$ by a copy of the XML forest F_i. Hence, if $\widehat{I}$ and $\widehat{J}$ have homomorphically equivalent finitely branching limits, then so do I and J on all possible finite inputs. □

Theorem 2 can be intuitively rephrased as follows: when 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 those with input.

5.1. Query languages

The query languages we consider are all variants of tree pattern queries [6]. They are different in the use of joins and/or constructors that is allowed.

TPQs with Joins. We first define 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.

This semantics can be straightforwardly extended to a *set* of XML documents. This extension is required in our setting, since in an AXML system a query is in general applied to more than one XML document. Let q be a TPQ-J and $\{I_i | i = 1, \ldots, n\}$ a set of XML documents. Evaluating q on $\{I_i\}$ yields the union of the results of evaluating q on each of the documents (i.e., $\bigcup_i \{q(I_i)\}$).

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.

Definition 10 (Tree Pattern Queries with XPath-Joins). *A* tree pattern query 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.*

Definition 10 is arguably somewhat involved. However, the condition (†) ensures that all TPQ-XJs are expressible in XPath using data equality tests, and *this is the only fact about TPQ-XJ that will use in what follows.* More precisely, let $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ be the fragment of navigational XPath studied in [10], in which only the downward axes $\downarrow$ and $\downarrow^+$ may be used and in which data equalities are allowed as tests. We refer to [10] for the precise syntax and semantics. Then we have the following.

Proposition 5. *Every TPQ-XJ can be translated in polynomial time to an equivalent $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ path expression.*

Proof. We give the basic outline of the translation. The details are straightforward, but we omit them because they would require us to go into the precise syntax and semantics of $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ [10], which is not relevant for the rest of the paper.

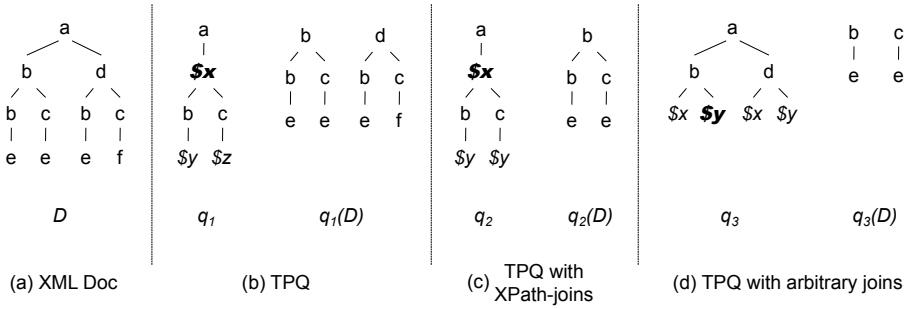


Figure 7: Examples of TPQs with joins

Consider any TPQ-XJ. By induction, we can associate to each node of the query either

(i) an $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ *path expression* (if the node is an intermediary, or if the node lies on the path from the root to the result node), or

(ii) an $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ *node expression* (otherwise).

The $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ expression associated to a leaf is either $.[p]$ (if the leaf is labeled by some $p \in \mathcal{L}$, or simply . (if the leaf is labeled by an element of $\mathcal{V}$). The $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ expression associated to a non-leaf node is obtained by combining the $\mathsf{XPath}(\downarrow,\downarrow^+,=)$ expressions associated to the children of the node, as well as the label of the node in question. Data equalities are used whenever the node in question is the least common ancestor of a pair of nodes labeled by the same variable.

Since the root trivially lies on the path from the root to the result node, we obtain, in the end, a path expression that, when evaluated at the root of a document, yields precisely the nodes returned by the TPQ-XJs. □

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. Edges of the form | are child edges. A query can also contain descendant edges denoted by ||, as for instance in the query corresponding to node q in the system of Figure 1. 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 to meet both subconditions of* (†).

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.

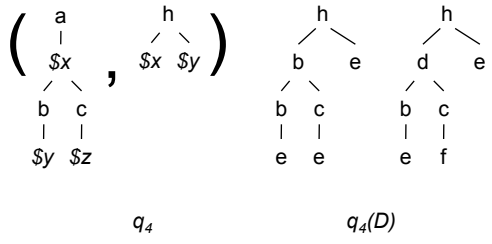


Figure 8: Example of TPQ with constructors

Definition 11 (TPQs with Constructors). *A tree pattern query with constructors (TPQ-C) is a pair (q, t) 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.*

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 unique node of q labeled x. Note that, here, we use the fact that q is a TPQ, as opposed to an arbitrary TPQ-J, because in TPQs each variable may occur only once. In principle, the definition could be adapted to also incorporate joins, but it will turn out that the use of TPQ-Cs quickly leads to undecidability even without joins, and therefore the above definition will be sufficient. Also, note that the result node of q plays no role in this definition.

Just as in the case of TPQ-Js, a TPQ-C may be applied to a *set* $\{I_1, \ldots, I_n\}$ of XML documents. In this case, $q(\{I_1, \ldots, I_n\}) = \bigcup_{1 \leq i \leq n} \{q(I_i)\}$.

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

5.2. Input-free Systems

Recall from Section 4 that, for a query-free system I, we were able to construct a finite *graph* G_I whose unraveling is a finitely branching limit of I. Using these graphs, and using the concept of *simulations*, we were then able to test the equivalence between any two query-free systems.

For input-free systems with queries, a graph representing a limit of the system cannot be obtained as easily from the system as before, because, intuitively, computing the limit involves repeatedly evaluating the queries of the system. However, we will show that for *input-free* systems with TPQ-Js, a Datalog program can be constructed that computes such a graph. If the system uses only TPQ-XJs, 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. The same strategy works for systems with arbitrary TPQ-Js, but with a slightly higher computational complexity. 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. Let $I = (N, E, \lambda)$ be an input-free system containing TPQ-Js. We will specify the Datalog program Π_I that computes as its output a graph that represents a limit of the system. The Datalog program will contain a constant n for each node n of I, as well as a constant a for each node label $a \in \mathcal{L}$ occurring in I. We will use $X, Y, Z, \ldots$ as variables.

child(m,n) :- (for $(m,n) \in E$)
label(m,a) :- (for $\lambda(m) = a \in \mathcal{L}$)
child(X,Y) :- child(X,m), child(m,Y) (for $\lambda(m) = rcv_i$ and $\lambda(n) = snd_i$)
child(X,Y) :- child(X,m), child(m,U), q(U,Y) (for $\lambda(m) = q$)
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 written out as a conjunctive query using the child, desc and label relations, where X is identified with the root, and Y with the result node of q. The intuition behind this Datalog is as follows: the first two rules (or, more precisely, the first two sets of rules, with are parametrized by m, n and a) encode the basic tree-structure of the AXML system. The third rule captures the semantics of send and receive nodes. The fourth rule captures the semantics of query nodes. The last two rules, finally, are simply bookkeeping rules that compute the descendant relation, as the latter may be used inside the queries q.

The Datalog program Π_I computes as its output, in a natural way, a graph G_I. More precisely, G_I is the graph (N', E', r, λ), where $N' \subseteq N$ is the set of all passive nodes of I, $E' \subseteq N' \times N'$ is the child relation over N' computed by Π_I, r is the root node of I, and λ is the labeling function of I restricted to N'.

In this way, we get an analogue of Proposition 3 for AXML systems with TPQs with joins:

Proposition 6. *For every input-free system with TPQs with joins, $unr(G_I)$ is a finitely branching limit of I.*

Proof. The proof is along the same lines as that of Proposition 3. In part (i) of the proof, there is an additional case, where a query node is invoked, but the argument used there is essentially the same as in the case of a send node invocation.

In part (ii), note that if a node n'' is a child of a node n' in $unr(G_I)$, there are three possibilities: (a) there is an edge from $\pi(n')$ to $\pi(n'')$ in I; (b) the edge from $\pi(n')$ to $\pi(n'')$ was added to G_I because n' has a receive node child and n'' has a send node parent, and there is a chain of communication channels connecting the channels of the send and the receive node in question; (c) the edge from $\pi(n')$ to $\pi(n'')$ was added to G_I because $\pi(n')$ has a query node as

a child, and n'' is among the answers of the query in question. The first two cases were already dealt with in the proof of Proposition 3. For third case, note that if $(n, m) \in q(G_I)$ (i.e., if m is among the answers of the query q when evaluated from node n), and $unr(G_I)_{n'}|_k \xrightarrow{\text{hom}} (I'_{\downarrow})_n$ for any node n' such that $\pi(n') = n$ and for big enough k, then it follows from the induction hypothesis that $m \in q(content(n, I'))$ and hence, if I'' is the AXML system obtained from I' by invoking the send node in question, then the node n in I'' has a child that is a copy of m. □

Using Propositions 1 and 2 immediately yields an algorithm for testing equivalence of two input-free systems with TPQs with joins. In particular, according to Proposition 1, checking equivalence of two input-free systems with finitely branching limits reduces to checking homomorphic equivalence of their limits. In the case of an input-free system I with TPQs with joins a finitely branching limit is $unr(G_I)$, as shown by Proposition 6. Thus to check equivalence of two such systems I and J it suffices to check for the homomorphic equivalence of $unr(G_I)$ and $unr(G_J)$, which according to Proposition 2 can be done by checking whether G_I simulates G_J and vice versa. The only piece missing for deciding the equivalence of two such systems is constructing G_I (resp. G_J), which can be done by executing a Datalog program, as explained above.

Since this 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 $\text{P}_{||}^{\text{NP}}$, i.e., deterministic polynomial time with parallel access to an NP-oracle [15]. This follows, as we will explain, from the fact that the arity of the relations in the constructed Datalog program is bounded. In fact, we have the following:

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

PROOF. For the upper bound, it suffices to show that the Datalog program Π_I can be executed in time $\text{P}_{||}^{\text{NP}}$. The remainder of the algorithm consists of testing whether the resulting graphs for the two systems simulate each other, which by Proposition 2 can be done in polynomial time.

Observe that the arity of the relations in the Datalog program Π_I is bounded by a constant where the constant in question is 2). The combined complexity of evaluating a Datalog program whose relations have bounded arity is in $\text{P}_{||}^{\text{NP}}$. To see this, first observe that there are only polynomially many possible atomic facts using constants and relations occurring in the program (where the degree of the polynomial depends on the maximal arity of the relations). Hence, if a fact belongs to the answer of the program, this can be witnessed by a polynomial size derivation, and therefore the problem of testing whether a fact belongs to the answer of the program is in NP (we assume that "polynomial size derivation" includes a homomorphism witnessing each application of a rule). The complete answer of the Datalog program can then be computed by asking an NP-oracle, for each of the polynomially many possible facts (in parallel), whether or not it belongs to the answer of the Datalog program.

For the lower bound, we will first define another problem and show it to be $\text{P}_{||}^{\text{NP}}$-complete. The problem is:

($*$) *given two unary conjunctive queries and an instance, decide if the two queries have the same answers on this instance.*

This problem is clearly decidable in $\text{P}_{||}^{\text{NP}}$ (the algorithm uses an NP-oracle to test, for each element of the active domain of the instance, whether it belongs to the answer of each of the two queries, and, depending on the result, outputs *yes* or *no*). The $\text{P}_{||}^{\text{NP}}$-hardness of ($*$) is proved by a reduction from the following problem, which is known to be complete for this complexity class [17]: *given two graphs, decide if they have the same chromatic number.* For any $k \geq 1$, let I_k be the instance (for a schema consisting of a single binary relation) that is a disjoint union of cliques of sizes $1 \ldots k$. For any graph G, let q_G be the canonical conjunctive query of G, i.e., the Boolean conjunctive query whose existential variables are the elements of G and whose conjuncts are the edges of G. Then it is not hard to see that two graphs G, G' have the same chromatic number if and only if q_G and $q_{G'}$ have the same answers on the instance I_k, where k is the maximum of the number of vertices of G and G'. Note that this reduction uses only a single binary relation, and hence, the problem ($*$) is already $\text{P}_{||}^{\text{NP}}$-complete for queries and instances over a fixed schema consisting of a single binary relation.

Finally, we show how to reduce ($*$) to the problem at hand. We assume a fixed schema consisting of a binary relation R. We can associate to each instance $I = \{R(a_1, b_1), \ldots, R(a_n, b_n)\}$ an XML tree

$$t_I = root\{R\{1\{a_1\}, 2\{b_1\}\}, \ldots, R\{1\{a_n\}, 2\{b_n\}\}\} \ .$$

Analogously, we can associate to each unary conjunctive query q a TPQ-J q' (over XML trees): for each conjunct $R(u, v)$ of q, q' contains a subtree below its root of the form $R\{1\{\$u\}, 2\{\$v\}\}$. Then two unary conjunctive queries q_1, q_2 have the same answers on an instance I if and only if the AXML system $root\{q_1'\{t_I\}\}$ is equivalent to the AXML system $root\{q_2'\{t_I\}\}$. □

Since NP is contained in $\text{P}_{||}^{\text{NP}}$, in particular, this shows that testing equivalence of AXML systems with TPQs with joins is NP-hard.

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 [7] that XPath expressions, seen as conjunctive queries, have *bounded tree-width*. Recall from Proposition 5 that every TPQ-XJ is equivalent to an XPath expression. Hence, every TPQ-XJ, viewed as a conjunctive query, has bounded tree-width. It follows that the constructed Datalog program Π_I has bounded tree-width as well, in the sense that the body of each rule is a conjunctive query of bounded tree-width. Without going into the definition of tree-width, what matters here is that the combined complexity of evaluating a conjunctive query is known to be in PTIME if the conjunctive query has bounded tree-width [11] (cf. [9] for more details). It follows, that evaluating a Datalog program of bounded tree-width

whose relations have bounded arity, is also 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 TPQs 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.*

PROOF. It can be proven by reduction from the acceptance problem of Turing Machines (TMs). This is based on the observation that any TM can be simulated by an AXML system with TPQs with constructors. Given a TM M and an input T, we construct two AXML systems I_1 and I_2. I_1 simulates M on T and outputs a fresh symbol *accept* iff M accepts the input. I_2 is simply the system $root\{accept\}$. Given I_1 and I_2, as described above, it is easy to see that I_1 and I_2 are equivalent iff M accepts input I. A configuration of M (i.e., its state and input tape) is modeled as a tree. Each transition is simulated by a query node that checks whether the configuration satisfies the transition condition and, if it does, outputs the new configuration. To make sure that the query nodes implementing the transitions operate on each configuration that M goes through, the output of each query node is sent to an internal channel and recursively received by children of the query nodes. Finally, another query node outputs *'accept'* when M reaches its final state.

More precisely, the system I_1 simulating M on T consists of:

- a *root* node with children
  - a send node $send_1$ with children
    - one query node q_i for every pair of transition rule and symbol in the alphabet of M with child a rcv_1 node
    - the encoding of the initial configuration, constructed as described below
    - a query node $q_{extend-left}$ with a child rcv_1 node
    - a query node $q_{extend-right}$ with a child rcv_1 node
  - a send node $send_2$ with a child query node q_{final} that has a child rcv_1 node
  - a rcv_2 node

In this construction, a configuration (s, t) of M, where s is the current state, and $t = x_n \dots x_1 h y_1 \dots y_m$ the input tape with h being the symbol under the head, is encoded as:

$$\{\boldsymbol{s}\{\quad h\{\quad \boldsymbol{l}\{x_1\{x_2 \ldots \{x_n\{\#\}\}\}\}\},\quad \boldsymbol{r}\{y_1\{y_2 \ldots \{y_m\{\#\}\}\}\}\quad\}\quad\}\}$$

where $\#$, l and r are fresh symbols not contained in M's alphabet.

Each q_i is a TPQ with constructors that implements one of the transition rules of the TM for a particular symbol in M's alphabet. For instance, consider the transition rule $(\sigma, s) \to (\sigma', s', RIGHT)$, specifying that on symbol σ and state s, the TM M replaces σ by σ', transitions to state s' and moves one position to the right. This transition rule can be implemented by the following class of TPQs with constructors; one for each symbol c in M's alphabet: (q, t), where $q : \{\boldsymbol{s}\{\sigma\{\boldsymbol{l}\{\$x\}, \boldsymbol{r}\{c\{\$y\}\}\}\}\}$ and $t : \{\boldsymbol{s'}\{c\{\boldsymbol{l}\{\sigma'\{\$x\}\}, \boldsymbol{r}\{\$y\}\}\}\}$. The query q_{final} checks if the current configuration is in the final state and if this holds, it outputs *accept*. Finally, $q_{extend-left}$ and $q_{extend-right}$ simulate the infinite tape by extending the tape with a blank symbol, whenever encountering the $\#$ symbol on the left and right end of the tape, respectively. □

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

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 this representation to also contain 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", i.e., the system contains only queries that operate directly on data received on external input channels, as opposed to operating on data generated by the system itself. An example of an AXML system that is *not* simplified is given in Figure 9a. Note that the query q_1 in this system operates on data received not on an external input channel but on an internal channel, while the query q_2 operates not on data received on an external input channel but on data actively created by part of the system. 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. This is because arbitrary AXML systems can combine queries with recursion (via the use of send and receive nodes), while simplified systems, in which all queries must operate on external input data, cannot apply queries

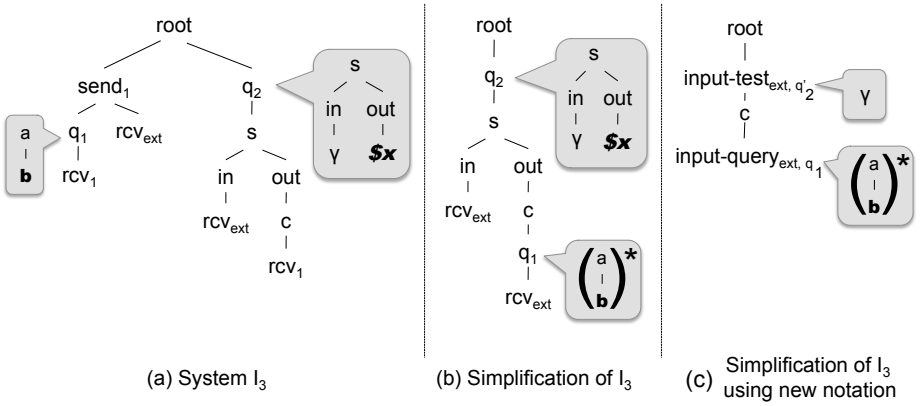


Figure 9: Example of system simplification

recursively. To this end we generalize the query languages we use, by allowing a limited form of recursion inside the queries themselves.

Definition 12. *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, the additional expressive power of RTPQs will 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 external inputs).

Before we proceed to define simple systems, we give an example illustrating why it will be important to allow RTPQs when we define simple systems.

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. According to the definition given below, the system I is not simple, because the query q_1 is applied over an internal channel. However, it is equivalent to the system $\{root\{q\{rcv_{ext}\}\}\}$, where q is the RTPQ $/(a/b)^*$. According to the definition below, the latter system is indeed simple. 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. Intuitively, 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 (i.e., perform a Boolean test 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 pieces of the data received on the external input channel* ext *to the output. On the other hand, q_2 uses the input differently. It 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 the following two types of queries: *input-query*$_{c,q}$ and *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, can be viewed as shorthand for $q\{rcv_c\}$. In other words, it corresponds to copying part of the input to the output. The second is allowed to have any number of children, and *input-test*$_{c,q}$ tests whether the query q has a non-empty result on the input received from input channel c. If the answer is positive, then a snapshot of the entire subforest below the the *input-test* node is copied appended as siblings of the *input-test* node. In other words, *input-test*$_{c,q}$ acts as a filter, and *input-test*$_{c,q}\{forest\}$ can be viewed as shorthand for $q'\{s\{in\{rcv_c\}, out\{forest\}\}\}$, where q' is the query

, which tests whether q has a non-empty result in the subtree below the *in*-node, and then returns all subtrees below the *out*-node.

Finally, we can define what it means for a system to be simple. We call an AXML system *simple* if it only uses queries in these two ways, i.e., if it can be viewed syntactically as an AXML system using the active node labels $send_c$, rcv_c as usual, and, instead of arbitrary query nodes, the active node labels *input-query*$_{c,q}$ and *input-test*$_{c,q}$ where c is an external input channel.

Example 10. *Figures 9c shows a simple system equivalent to I_3, using input-query*$_{c,q}$ *and input-test*$_{c,q}$ *nodes. When, as discussed above, input-query*$_{c,q}$ *and input-test*$_{c,q}$ *are viewed as shorthand notations, the system in Figure 9c unfolds into the system in Figure 9b.*

Making systems simple. For technical convenience, we will next 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 $/i$ (for $1 \leq i \leq n$) 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, which is spelled out in the Appendix, involves a detour through monadic Datalog [12]. More specifically, we identify a class of monadic Datalog 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.*

We describe the proof of 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 $\{send_1\{q_a\{rcv_2\}\}, rcv_c\}, send_2\{q_b\{rcv_1\}\}, rcv_1\}$. It is clear that the AXML system I_A computes the query $(q_a q_b)^*$, and therefore we can substitute any occurrence of the query by a copy of this AXML system.

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 other things, systems with TPQs to express regular path languages by simulating finite state automata.

5.4. Testing the Equivalence of Simple Systems with Input

It follows from results in [10] that the containment problem for unions of RTPQs with XPath-joins is decidable. In fact, a slightly more general result holds:

Proposition 7 ([10]). *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 an 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$(\downarrow, =)$, which is the main logic studied in [10], via a polynomial-time translation. 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. The 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 belongs to X. Note that I^X and J^X are simple systems without *input-test* nodes 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-test* nodes, 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 is contained in $\bigcup_{q' \in Y} q'$ 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 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 until no further *input-query* children need to be added to the document.

Finally, let I, J be simple systems without *input-test* nodes, 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 this bound is tight. However, we know that the equivalence problem for systems with TPQs and input 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 [16] (cf. the proof of Theorem 7).

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. Each axiom states that two forests are equivalent, in the sense that one may be replaced by the other inside the context of a bigger AXML system, without affecting the semantics of the overall AXML system. We show that the 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 $send_c\{F_1, F_2\} = send_c\{F_1\}, send_c\{F_2\}$
ax2 $send_c\{F_1\}, a\{F_2\} = a\{send_c\{F_1\}, F_2\}$
ax3 $send_c\{F\}, rcv_c = send_c\{F\}, F$
if there is no other $send_c$ node in the AXML system
ax4 $send_c\{F\} = \epsilon$ if there is no rcv_c node in the AXML system
ax5 $send_c\{rcv_{c'}, F\} = send_c\{F\}$
if every rcv_c node in the AXML system has a $rcv_{c'}$ sibling
ax6 $send_c\{F\} = \epsilon$ 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

Here, the variables F, F_1, F_2 denote AXML forests. The equality sign in these axioms is used to indicate that, inside any XML system, a subforest as in the left-hand side of the axiom may be replaced by the subforest as described in the right-hand side of the axiom, *and vice versa*, without affecting the semantics of the AXML system in question. In other words, the axioms can be used as equivalence-preserving undirected rewrite rules. For example, the axiom ax1, as an undirected rewrite rule, allows us to take an AXML system containing a subtree as described by the left-hand side (where c is any channel and F_1 and F_2 are arbitrary AXML forests) and rewrite it by replacing this subtree by the right-hand side (where F_1 and F_2 remain the same) *or vice versa*. In other words, using ax1 as an undirected rewrite rule, a send node with more than one child may be split into two send nodes, and two sibling send nodes acting on the same channel may be merged into one send node.

Some of these axioms have side conditions, that must be satisfied by the entire AXML system in order for the axiom to be applied. The side conditions of the axioms ax6-ax8 require some explanation, as they refer to "inaccessible channels" and to "simulations" between channels. 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 $c_n = c$, and rcv_{c_1} occurs in the system in a place that is not in the scope of any *send*-node, and each $rcv_{c_{i+1}}$ occurs in the scope of some $send_{c_i}$-node. Intuitively, a channel is accessible if data sent on this channel will eventually affect the snapshot of the system. This explains axiom ax6. A *simulation* in an AXML system is a binary relation Z between channels such that whenever $(c, d) \in Z$ and the system contains a subtree of the from $send_d\{F(rcv_{d_1}, \ldots, rcv_{d_n})\}$ then there are channels $c_1, \ldots, c_n$ such that the system contains $send_c\{F(rcv_{c_1}, \ldots, rcv_{c_n})\}$ and $(c_i, d_i) \in Z$ holds for all $i \leq n$. Here, we use the notation $F(rcv_{d_1}, \ldots, rcv_{d_n})$ for a forest containing receive nodes $rcv_{d_1}, \ldots, rcv_{d_n}$ (with $n \geq 0$) and we use $F(rcv_{c_1}, \ldots, rcv_{c_n})$ to denote the same forest in which the receive nodes $rcv_{d_1}, \ldots, rcv_{d_n}$ have been replaced by $rcv_{c_1}, \ldots, rcv_{c_n}$. If there is a simulation Z such that $(c, d) \in Z$, then we say that c simulates d. Intuitively, if c simulates d, then all data sent on channel d is also sent on channel c. This explains axioms ax7-ax8.

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 either of the form rcv_c or of the form $send_c\{a\{rcv_{c_1}, \ldots, rcv_{c_n}\}\}$ where a is a single passive node, and, furthermore,

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 $send_c\{a\{rcv_{c_1}, \ldots, rcv_{c_n}\}\}$ 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 2. *Using the axioms ax1-ax5 as undirected rewrite rules, every system can be rewritten to a normalized system.*

Proof. The axiom ax1 is used (in the right-to-left direction) to make sure there is a single send node per channel, after the axiom ax2 has been used (in the right-to-left direction) 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 containing a single passive node, by introducing intermediate channels. For instance, if a and b are passive labels, then using first ax4 and then ax3 (both in the right-to-left direction), $a\{b\}$ is rewritten to $a\{rcv_c, send_c\{b\}\}$. Finally, ax5 (in combination with ax2 and ax3) is used to ensure guardedness, i.e., that every rcv-node is directly below a passive node and not directly below a *send*-node. □

Notice that the definition of *simulations* that we gave above can be simplified when we consider normalized systems: a simulation in a normalized system is a binary relation Z between channels such that whenever $(c, d) \in Z$ 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 $send_c\{a\{rcv_{c_1}, \ldots, rcv_{c_n}\}\}$ and $(c_i, d_i) \in Z$ holds for all $i \leq n$. We say that a normalized system 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 3. *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 4 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 iff 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 and J are equivalent AXML systems. Let I' and J' be minimized normalized AXML systems such that I and I' are provably equivalent and J and J' are provably equivalent. Then I' and J' coincide (recall that we identify AXML systems up to isomorphism). □

In the case of acyclic systems (systems where the dependencies between the channels do not induce a cycle), fewer axioms are needed.

Theorem 11. *Two acyclic AXML systems are equivalent if and only if one can be rewritten to the other using the axioms ax1-ax4 together with*

$$ax9 \quad send_c\{\epsilon\} = \epsilon$$

as undirected rewrite rules.

PROOF. The axioms allow us to effectively eliminate all *send*-nodes and *rcv*-nodes for internal channels from an acyclic system. As before, using ax1 and ax2 we can rewrite any system into one in which there is only a single send node for each channel. This transformation preserves acyclicity. Furthermore, using ax9 we can make sure that whenever the system contains a receive node for some internal channel, then it also contains a send node for the same channel. Next, since the system is acyclic, all receive nodes for internal channels can be removed, one by one, using ax3. Finally, ax4 is used to remove all send nodes.

All this means we have to consider systems with external receive nodes only. It follows by Theorem 2 that any two such systems are equivalent if and only if they are homomorphically equivalent, i.e., they are isomorphic when reduced. Since we identify systems up to homomorphic equivalence, this gives us the result. □

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 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 system) [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".

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

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. Abiteboul, B. ten Cate, and Y. Katsis. On the equivalence of distributed systems with queries and communication. In *ICDT*, pages 126–137, 2011.

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

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

[8] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In J. E. Hopcroft, E. P. Friedman, and M. A. Harrison, editors, *STOC*, pages 77–90. ACM, 1977.

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

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

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

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

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

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

[15] K. Wagner. Bounded query classes. *SIAM J. Comput.*, 19(5):833–846, 1990.

[16] K. Wagner and G. Wechsung. *Computational Complexity*. D. Reidel Publishing Company, 1986.

[17] K. W. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theoretical Computer Science*, 51(1-2):53 – 80, 1987.

Appendix A. Proof of Theorem 6 (simplifying systems with queries and input)

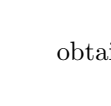
This appendix is dedicated to the proof of Theorem 6. To simplify the presentation, we focus on the case of TPQs without joins. As we will explain, the proof extends with minor modifications to TPQs with XPath-joins (but not to TPQs with arbitrary joins).

TP-Datalog. We start by defining a fragment of monadic Datalog with the same expressive power as RTPQs. We call it TP-Datalog, where TP stands for "Tree Pattern". The idea is that the body of each rule is a tree pattern (or, rather, a forest pattern, since we do not require it to have a single root).

Definition 13 (Forest-pattern query). *Forest-pattern queries (FPQs) are defined in the same way as tree pattern queries, except that the existence of a unique root is not required (i.e., there may be more than one root node), and there is no designated result node (and therefore, FPQs are Boolean queries). Forest-pattern queries with XPath-joins (FPQ-XJs) are defined in the same way, where condition (†) is required to hold if a root were added on top of the forest, and this root were the result node.*

An example of an FPQ is

. An example of a FPQ-XJ is 

. Note that this is indeed a FPQ-XJ because if we add a single new root node above the other nodes, and if we designate this new node as the result node, we obtain the TPQ-XJ 

, which satisfies the (†) condition.

Definition 14 (TP-Datalog). *A TP-Datalog program is a monadic Datalog program whose EDBs are the binary relations "ch" (for the* child *relation) "desc" (for the* descendant *relation) and "label" (where label(x, y) represents the fact that the node x has node label y) and unary relation "root", consisting of rules that are of one of the following forms:*

$$Q(y) \leftarrow root(y)$$

$$Q(y) \leftarrow P(x), q(x, y)$$

$$Q(y) \leftarrow P(x), q(x, y), P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$$

with $x, y, x_1, \ldots, x_n$ are distinct variables, q a TPQ with root x and result node y, and q' is a FPQ with roots $x_1, \ldots, x_n$. TP-XJ-Datalog programs are defined similarly, but using TPQ-XJs and FPQ-XJs instead of TPQs and FPQs.

Proposition 8. *Every TP-Datalog query is equivalent to a disjunction of queries, each of which is the conjunction of an RTPQ with zero or more Boolean RTPQs. The same holds for TP-XJ-Datalog and RTPQ-XJs.*

PROOF. We present the argument in depth for the case of TP-Datalog, and then explain how it extends to TP-XJ-Datalog.

Let (Π, P_0) be a TP-Datalog query (i.e., Π is a TP-Datalog program, and P_0 a designated IDB predicate defining the query). First suppose that all rules are of the form $Q(y) \leftarrow root(y)$ or $Q(y) \leftarrow P(x), q(x,y)$ with q a tree-pattern query whose root is x and whose result node is y. In other words, the rule bodies do not contain any "root-labeled forest pattern queries" of the form $P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$. In this case, it is easy to construct for each IDB P of Π an equivalent RTPQ, by viewing Π as a non-deterministic finite state automaton (NFA). Each IDB corresponds to a state of the NFA, and each rule constitutes a transition. The IDBs Q for which the program contains a rule $Q(y) \leftarrow root(y)$ are the initial states of the NFA. We then apply the Kleene translation from NFAs to regular expressions in order to obtain an equivalent RTPQ.

The general case requires more work. Let $\mathcal{Q}$ be the set of all "root-labeled forest pattern queries" of the form $P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$ occurring in the rule bodies in Π. Recall that these are Boolean queries. There is a natural technique for eliminating the queries inside the Datalog program, which involves considering all possible ways in which a set of Boolean queries from $\mathcal{Q}$ may become true, one after the other, during the evaluation of the Datalog program. In other words, we consider all possible ordered subsets $\vec{X}$ of $\mathcal{Q}$. For each such ordered subset, we construct a program $\Pi^{\vec{X}}$ that simulates executions of Π assuming that, during the execution, the Boolean queries in $\vec{X}$ become eventually true, *in the order in which they are listed in* $\vec{X}$. It is easy to construct the program $\Pi^{\vec{X}}$ in question: if $|\vec{X}| = n$, then the program contains n copies $P^1 \ldots P^n$ of each IDB Π of Π, where, intuitively, P^k computes the extension of P after the first $k-1$ Boolean queries in $\vec{X}$ have become true. If Π contains a rule of the form

$$P_i(y) \leftarrow P_j(x), q(x,y)$$

or

$$P_i(y) \leftarrow P_j(x), q(x,y), P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$$

and the root-labeled forest pattern query $P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$ is among the first k queries listed in $\vec{X}$, then $\Pi^{\vec{X}}$ contains, for all $k \leq \ell \leq n$ the rule

$$P_i^{(\ell)} \leftarrow P_j^{(\ell)}(x) \wedge q(x,y)$$

reflecting the fact that, from stage k onwards, we assume the Boolean query $P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n)$ to be true.

Similarly, if Π contains a rule of the form $Q(y) \leftarrow root(y)$, then $\Pi^{\vec{X}}$ contains the rule $Q^{(k)}(y) \leftarrow root(y)$ for all $1 \leq k \leq n$.

We explain next how to express by means of a Boolean RTPQ that the ith Boolean query in $\vec{X}$ indeed evaluates to true given that the ones before did (this allows us to ensure that the assumption that the queries in $\vec{X}$ become true one after the other is warranted). Let this Boolean query be of the form

$$P_1(x_1), \ldots, P_n(x_n), q'(x_1, \ldots, x_n) \ .$$

For each $j \leq n$, we compute the RTPQ corresponding to $P_j^{(i-1)}$ (as described above via NFAs). Then we split q' into individual tree pattern queries rooted by $x_1, \ldots, x_n$ respectively, compose each tree pattern with the relevant RTPQ that we just computed, and take the big conjunction, obtaining a conjunction of Boolean RTPQs.

The final query we obtain is a big disjunction (corresponding to all possible ordered subsets of $\mathcal{Q}$), where each disjunct is a conjunction consisting of a Boolean RTPQ (which simulates the original query under the assumption that the chosen ordered subset of $\mathcal{Q}$ is correct) with Boolean RTPQs (which justify the assumption that the chosen ordered subset of $\mathcal{Q}$ is indeed consistent).

Finally, let us mention that, in the case with XPath-joins, the construction is essentially the same, but the step of "splitting of the forest-pattern query" is a bit more subtle. Here, the forest-pattern query q' cannot be decomposed into individual tree-pattern queries because there may be equality-edges linking nodes in different tree patterns. Nevertheless, it is easily seen that the structural property (†) of XPath-joins enables us to construct the desired Boolean RTPQs (Note that the condition (†) implies that for every tree in the forest q' there is at most one other tree such that there is a join connecting nodes in the two trees – that is, one tree cannot be involved in two joins with other trees). □

This shows that, in constructing the simple system equivalent to a given system, we can use TP-Datalog queries instead of RTPQs in the *input-test* and *input-query* nodes (indeed, queries of the form as described in the statement of Proposition 8 can easily be expressed by simple AXML systems using only RTPQs, using *input-query* nodes for the unary RTPQs and *input-tests* for the Boolean RTPQs)

Inspection of the proof of Proposition 8 shows that the translation is single exponential.

Incidentally, a converse of Proposition 8 hold as well, as can easily be seen by encoding RTPQs as finite state automata, and translating the latter to TP-Datalog (using a unary predicate per state of the automaton).

The graph of a system with input. Recall the definition of the graph associated to an input-free system. A similar graph can be constructed for a system with input. For simplicity, we consider the case where there is a single input channel and a single input tree. For each pair (I, T), where I is a system and T is an input tree, we construct a graph $G_{I,T}$ as follows: the nodes of the graph are the passive nodes of I, together with the nodes of T. There is an edge between two nodes n, m if, in the limit, every copy of n will have a child that is a copy of m. It can be seen that such a finite graph representing the limit of the system can always be constructed (indeed, one way to see this is to imagine the input as part of the system, sitting underneath some send-node, and using our previous results concerning input-free systems).

We will show how to construct, for every system I, a monadic Datalog program that takes as input a tree T and computes the graph $G_{I,T}$. Afterwards, by analyzing the datalog program further, we can obtain from it a way to turn

every system into a simple one. Before we can give the definition of the Datalog program, we need to introduce some auxiliary concepts.

Partial matchings. Let I be an AXML system, n a node of I, and q a TPQ. By a *partial matching* for q at n we will mean a partial map f from nodes of q to nodes of I, such that (i) the domain of f is prefix-closed, i.e., if a node belongs to the domain of f, then all its ancestors do too, (ii) f maps the root of q to n, and (iii) f preserves node labels (but not necessarily child and descendant edges). Intuitively, the idea of a partial mapping is that the nodes *not* in the domain of f are to be mapped to parts of the input data.

Given a partial matching f for a TPQ q at a node n in a system I, the *image* of q under f, denoted by $img(q, f)$, is the conjunction of all formulas of the form

- $ch_{f(x),f(y)}$ for $child(x, y) \in q$ with $x, y \in dom(f)$
- $desc_{f(x),f(y)}$ for $desc(x, y) \in q$ with $x, y \in dom(f)$
- $ch_{f(x)}(y)$ for $child(x, y) \in q$ with $x \in dom(f)$, $y \notin dom(f)$
- $desc_{f(x)}(y)$ for $desc(x, y) \in q$ with $x \in dom(f)$, $y \notin dom(f)$
- $ch(x, y)$ for $child(x, y) \in q$ with $x, y \notin dom(f)$
- $desc(x, y)$ for $desc(x, y) \in q$ with $x, y \notin dom(f)$
- $label(x, a)$ for $label(x, a) \in q$ with $x \notin dom(f)$

Intuitively, $img(q, f)$ is a conjunctive query listing the requirements that need to be satisfied in order for the partial matching f to extend to a real matching of q in the graph $G_{I,T}$.

The definition of partial matchings and of $img(f, q)$ extend naturally to the case of TPQ-XJs.

Construction of the datalog program. Let I be a system with TPQs and with input, using only a single input channel. Let T be an XML tree (in general it can be a forest but we will consider the case of a single tree, for simplicity). Furthermore, assume the tree T is represented as a structure with binary relations "child", "desc", "label" and unary relation "root".

We will construct a TP-datalog program that takes as input the tree T and that computes the graph $G_{I,T}$ we described above. Note that I is fixed (whereas T is not.) The TP-datalog program uses many unary IDB relations, indexed by nodes of I. These relations compute for each node n of I, which nodes of T will eventually become children of n, and also which nodes of I will eventually become children of n, during execution of I.

For convenience, we will initially present the datalog program using unary and zero-ary IDB relations. Afterwards, it will be clear that the program can be equivalently written using only unary IDB relations, in which case it is indeed a TP-datalog program.

Let N be the set of all passive nodes of I. The datalog program consists of the following rules:

- $ch_{n,m} \leftarrow$; for all nodes $n, m \in N$ such that m is a child of n or m is a child of a send-node and n is the parent of a corresponding rcv-node.
- $desc_{n,m} \leftarrow ch_{n,m}$ for all nodes $n, m \in N$
- $desc_{n,m} \leftarrow ch_{n,n'}, desc_{n',m}$ for all nodes $n, m, n' \in N$
- $desc_n(x) \leftarrow ch_n(x)$ for all nodes $n \in N$
- $desc_n(x) \leftarrow ch_n(y), desc(y, x)$ for all nodes $n \in N$
- $desc_n(x) \leftarrow desc_{n,m}, desc_m(x)$ for all nodes $n, m \in N$

Finally, for each node n having a child m labeled by q, and for each partial matching f of q at a node n', we add

- $ch_{n,f(y)} \leftarrow ch_{m,n'}, img(f, q)$ if $y \in dom(f)$, or
- $ch_n(y) \leftarrow ch_{m,n'}, img(f, q)$ if $y \notin dom(f)$

When run on input T, the program indeed computes the graph $G_{I,T}$, in the following sense: for all nodes n, m of I, $ch_{n,m}$ computes given T whether n is a child of m in the graph $G_{I,T}$. Intuitively, the program searches in T and I for data that implies that an edge should exist between from n to m. Similarly, for each node n of I, the unary IDB relation ch_n computes the nodes in T that are the children of n in the graph $G_{I,T}$.

Now, remark that all zero-ary IDB relations can be replaced by unary ones (whenever the body of a rule contains a zero-ary relation B, it can be replaced by $B(x)$ with x a fresh variable, and whenever the head of a rule reads B, it can be replaced by $B(x)$ where x is a fresh variable, and $root(x)$ is added to the body of the rule in question. The reader may verify that, after this modification, the above program Π is indeed a TP-datalog program, and hence the $ch_{n,m}$ and ch_n relations are indeed TP-datalog queries.

Finally, given the TP-datalog program Π, and using Proposition 8, we can easily construct a simple system containing RTPQs that is equivalent to the original system I (the query computed by an IDB $ch_{n,m}$ of Π is used, after conversion to RTPQ, as an input-test, while the query computed by an IDB ch_n is used, after conversion to RTPQ, as an input-query).

The argument extends to the case where the input system contains TPQ-XJs, in which case the program constructed is a TP-XJ-Datalog program, and the final simplified system may contain RTPQ-XJs.

Complexity analysis. The construction of the TP-datalog program from the system is single exponential, due to the fact that a query, in general, has exponentially many partial matchings in a system. Converting the TP-datalog program into RTPQs may involve another single exponential blowup. Finally, converting the obtained graph with queries into a simple system can be done in PTIME. Hence, all in all, the algorithm transforming an arbitrary system with TPQs into a simple system with RTPQs runs in 2EXPTIME. The same complexity bounds are obtained in the case of TPQs with XPath-joins.