Stream Firewalling of XML Constraints

Michael Benedikt
Oxford University

Alan Jeffrey
Bell Labs, Alcatel-Lucent

Ruy Ley-Wild*
Carnegie Mellon University

ABSTRACT

As XML-based messages have become common in many client-server protocols, there is a need to protect application servers from invalid or dangerous messages. This leads to the *XML stream firewalling problem*; that of applying integrity constraints against a large number of simultaneous streams. We conduct the first investigation of a constraint engine optimized for the generation of XML stream firewalls. We isolate a class of DTDs and XPath constraints which support the generation of low-space filters, and provide algorithms for generating firewalls with low per-input-character time and per-stream space. We give experimental results which show that we have achieved these goals in practice.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query Processing

General Terms

Algorithms

Keywords

XML, XPath, Streams, Query Processing

1. INTRODUCTION

The *XML stream filtering* problem is a core technical challenge in XML-based middleware. From the work of Altinel and Franklin [1] onward, it has been a subject of intense research activity [6, 5, 12, 9]. In filtering one has a large collection of (user-supplied) XPath queries being executed on a single (system-supplied) streamed XML document – the goal of the middleware is to rapidly identify the elements within the stream that satisfy each query, with the goal of routing content to an appropriate endpoint. In this paper we propose the dual *XML stream firewalling* problem, where a single (system-supplied) XPath query is executed on a large number of (user-supplied) streamed XML documents – the goal of the middleware is to rapidly identify XML documents which violate the query, with the goal of rejecting documents based on security policy.

A typical XML stream firewalling scenario is a server exposing a web service interface to a large number of clients. For example, consider a web service receiving authenticated GET and PUT messages, modeled on HTTP:

```
<messages>
  <message type="get" href="foo">
    <head>
      <userid>12345</userid>
      <challenge>abc123def</challenge>
      <response>123abc456</response>
      ...
    </head>
  </message>
  <message type="put" href="bar">
    <head>...</head>
    <body>...</body>
  </message>
    ...
</messages>
```

One set of constraints concern the security of the server itself. The majority of clients are honest agents and upload valid XML satisfying appropriate data integrity constraints. However, the server must be protected against against dishonest agents, which may attempt to exploit server vulnerabilities by sending crafted messages. For example, to defend against SQL injection attacks, we could filter incoming messages and reject userids containing single-quote characters, by applying the following constraint to the root node:

```
not(//head[contains(userid,"'")]
```

Other constraints may enforce conformance to the messaging protocol, allowing the server to focus on message sequences that are reasonably well-behaved. For example, one may want to enforce that any message containing a response must contain a challenge:

```
not(//head[response and not(challenge)])
```

or that each message has at most one userid:

```
not(//head/userid/preceding-sibling::userid)
```

The last two constraints could be embodied in the DTD for the web service (at the cost of increasing complexity of the

*This material is partly based upon work supported by the National Science Foundation under Grant No. 0430175.

DTD) but some constraints cannot, such as a requirement that GET requests have no body:

```
not(//message[@type="get"]/body)
```

Although it is possible to interleave integrity constraint-checking with web service application code, a more modular design is to have a separate *application-layer firewall* dedicated to constraint-checking. In our work we will consider firewalls that enforce integrity constraints given in a subset of XPath that includes filters, boolean operations, and both vertical (parent/child) and horizontal (sibling) navigation. We will also consider traditional schema constraints, given as Document Type Definitions (DTDs). Together, these give a powerful constraint specification mechanism.

In common web service examples, the underlying communication protocol (such as TCP/IP) fragments each XML stream into packets. Servers will often have large numbers of simultaneous streams receiving interleaved fragments of XML. For each XML fragment, an application layer firewall must restore the current state of the stream, process the fragment, and then save the updated state and forward the fragment (as long as the constraint has not been violated). Since there may be large numbers of simultaneous streams, the per-stream space usage of the firewall is of great importance.

Moreover, it is important that a firewall detect attacks as early as possible. In the above example, a server may be processing messages as they arrive, and so it is important that a firewall reject an XML fragment as soon as a constraint violation is detected: waiting for the closing `</messages>` will be too late. Indeed, due to interleaving of streams as described above, even a small delay in detection can result in many streams being processed while the server is in an unsafe state. We call this the *fast-fail* requirement.

Another requirement for XML stream firewalls is that it is not possible to mount denial-of-service attacks on the firewall itself. In particular the per-input-stream space must be fixed, as otherwise an attacker could craft a message stream to cause the firewall to exhaust its memory. In summary, our priorities are:

- low per-input-character time,
- low, fixed per-input-stream space, and
- fast-fail solutions.

The requirement of bounded per-input-stream space means that the depth of the input document must be bounded. Without this restriction, an attacker could open a large number of streams containing data such as:

```
<html><body><i><b><i>...
  // long sequence of random <i> and <b> tags
```

Any validator processing documents of this form will have to store a stack linear in size to the input document, which will make it simple for an attacker to exhaust the validator's memory. Fortunately, XML-based message formats often do not normally require unbounded documents. To achieve a fixed per-input-stream footprint, we will require a nonrecursive DTD, against which input documents will be validated by the firewall. This restriction ensures that the XML stream firewall can be implemented as a deterministic finite automaton (DFA), rather than either a finite state automaton with a separate stack controller as in [11, 12] or the pushdown transducers required by [16, 17]. The documents for our example above might satisfy the nonrecursive given in Figure 1.

We will also make restrictions on the XPath constraints, to guarantee that runtime space usage is low:

- we exclude *data joins*
- we exclude *rightward moves*, either explicitly (through axes such as following-sibling::B) or implicitly (through paths such as ancestor::A/descendant::B), and
- we require all axes to be *named* (allowing, for example, ancestor::A but not the wildcard ancestor::$*$).

We will show that this sublanguage, which we denote *efficiently streamable XPath*, ensures that the XML stream firewall can be implemented with run-time space that is not only independent in the stream, but *linear* in the size of the constraint. In Section 7, we give experimental evidence to suggest that these restrictions do not dramatically impact expressiveness: any XPathMark [10] query which could be implemented as a DFA was expressible in our XPath fragment. Note that these constraints permit arbitrary boolean operations, possibly nested; the use of negation and disjunction is rare in XML filtering, but essential in XML constraint processing, since constraints specify exactly what must *not* happen in a message.

Existing work on XML stream processing focuses on the dual case of applying large numbers of queries on a single stream. One common approach is to generate an automaton that does the enforcement – usually a finite-state automaton that is coupled at runtime with a fixed PDA. [1] compiles into a set of automata, while [9] compile into a single nondeterministic automaton, optimized to exploit the sharing among queries. The disadvantage of these approaches is that the complexity of a flat automaton, even a nondeterministic one, can be exponential in the size of a filter. NFA approaches also do not deal well with negation (which often plays a crucial role in constraints) as complementation of an NFA can lead to an exponential blow-up.

An alternative is run-time generation of an automaton [11], in which states of a (in [11], deterministic) automaton are generated on-demand. The worst case bounds for this "lazy automaton" approach are still exponential, as in the case of pre-compiled automata. Furthermore, the run-time approach is more limited in what kinds of static pruning can be done in advance. However, there are advantages of the run-time approach in the context of the filtering problem. It can react to the addition of new filters without expensive recompilation, and can exploit regularities in the data that are not captured in a DTD. Indeed [11] shows that the size of the runtime automata can be large only if the runtime data is itself complex. It is easy to see that the runtime approach is not a good match for the firewalling problem. In our context we have the ability to do static optimization, since our constraints are generally known in advance. In firewalling we wish to do optimization in order to *fail fast* – to abort processing of a corrupt message at the earliest possible time. Finally, since the goal is to deal with large numbers of users that may not conform to the constraints, or even the DTD, we can not assume that most users will behave in a regular way. As with all security applications, once attackers discover our worst case, it will soon become our average case.

Our point of departure will be a third approach to XML filtering, that of *transducer networks*. Transducer networks are a "symbolic" representation of a finite state machine, that can be exponentially succinct compared to the corresponding NFAs. They can be constructed in time polynomial in the filter, and can be executed efficiently in term of space and time. They thus provide a streaming algorithm that is polynomial time in both filter and the data. As we will show here, the transducer network construction, when applied to efficiently streamable XPath constraints, yields finite state transducers of size linear in the constraint.

The main limitation of transducer networks is that they are difficult to statically analyze. In particular, the state pruning and path-sharing techniques that are available in automata-based approaches [9, 11], as well as the specialized data structures used there [6], are not applicable to transducer networks. Static analysis is important to XML stream firewalling in order to implement fast-fail firewalls. Our solution will be to look at another symbolic representation of automata.

We will focus on *Binary Decision Diagrams* [3] (*BDD*s) which are a common technique in symbolic model checking [8]. Prior work outside of the XML setting has shown that BDDs are promising for filtering [4] BDDs can be used to represent both XPath constraints and DTDs. They retain the advantages of explicit automata representations: fast evaluation and the ability to do simplification. In addition, we will show that XPath constraints and DTDs can be translated into BDDs with blow-up subquadratic in the size of the constraints. We show how simplification of BDD-based automata allows us to obtain fast-fail constraint processors.

We provide details of an implementation of a BDD-based XPath processor which can execute constraint-checks extremely rapidly, and compare the performance with state-of-the-art XPath filtering engines such as XMLtk [11], YFilter [9], XSQ [17], and GCX [19].

In summary, the contributions of this paper are:

- the first investigation of an XPath engine optimized for the generation of XML stream firewalls,
- the isolation of a fragment of XML/XPath which supports the generation of low-space filters,
- algorithms for generating transducer networks for enforcing such filters,
- the first use of BDDs as a runtime representation of XML constraints, and
- experimental results which compare performance with state-of-the-art XPath engines.

2. PRELIMINARIES

For any set Σ let Σ^* be the set of strings over Σ, and for any function $f : \Sigma \to \Delta$, let $f^* : \Sigma^* \to \Delta^*$ be its extension to strings. Let $\Sigma \times \Delta$ be the cartesian product of Σ and Δ with projection functions $\pi_1 : \Sigma \times \Delta \to \Sigma$ and $\pi_2 : \Sigma \times \Delta \to \Delta$. For any finite set Σ, write $|\Sigma|$ for the number of elements of Σ. For any string $s \in \Sigma^*$, write $|s|$ for the length of s.

By an *automaton* $\mathcal{A}$ over alphabet Σ we will always mean a finite automaton over words, that is a quadruple $(Q, \to, I, F)$ where Σ and Q are finite sets (the alphabet and state set respectively), $\to \subseteq Q \times \Sigma \times Q$ (the transition relation), and

```
<!ELEMENT messages (message*)>
<!ELEMENT message (head,body?)>
<!ELEMENT head ((userid|challenge|response)*)> ...
```

$$\begin{array}{rcl} \mathsf{messages} & \mapsto & \mathsf{message}^* \\ \mathsf{message} & \mapsto & \mathsf{head\ body?} \\ \mathsf{head} & \mapsto & (\mathsf{userid} \mid \mathsf{challenge} \mid \mathsf{response})^* \cdots \end{array}$$

Figure 1: Example DTD D_1

$I, F \subseteq Q$ (the initial and final states respectively). We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \to$. For $s \in \Sigma^*$, write $q \overset{s}{\Longrightarrow} q'$ for the transitive reflexive closure of $\to$, that is:

$$q \overset{s}{\Longrightarrow} q' \text{ whenever } q = q_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_n = q' \\ \text{and } s = a_1 \ldots a_n$$

The language induced by an automaton $\mathcal{A}$ is defined as usual. A *sink state* of an automaton is one which can not reach an accepting state. We write $|\mathcal{A}|$ for $|Q| + |\to|$.

We will deal with XML documents abstracted as *ordered trees* over some set of tags Σ. To simplify the presentation, we will not give an explicit treatment of attributes or PCDATA, and instead treat them as nodes like any other.

Stream processing will deal with the standard serialization of XML documents, as a sequence of begin and end tags. For a set of tags Σ, we let $\mathrm{Tags}(\Sigma) = \{\langle A \rangle, \langle /A \rangle \mid A \in \Sigma\}$ be the set of begin and end tags with labels in Σ. For an ordered tree T with node labels Σ, let $\mathrm{stream}(T) \in \mathrm{Tags}(\Sigma)^*$ be its textual representation.

In this paper, we will consider specifications for constraints using both schemas given as Document Type Definitions (DTDs) and application-specific constraints given in XPath.

A *DTD* D over Σ is an initial element $A_0 \in \Sigma$ together with a deterministic (i.e. 1-unambiguous) regular expression $D(A)$ over Σ for each $A \in \Sigma$. A DTD is nonrecursive if $\Sigma = \{A_1, \ldots, A_n\}$ where $A_i \in s \in \mathcal{L}(D(A_j))$ implies $i > j$. The definition of an ordered tree being validated by a DTD is standard, and we write $\mathcal{S}(D) \subseteq \mathrm{Tags}(\Sigma)^*$ for the set containing $\mathrm{stream}(T)$ for any T validated by D. We write $|D|$ for $|D(A_1)| + \cdots + |D(A_n)|$.

We will focus on XPath constraints that exclude data joins (sometimes called "Navigational XPath" [14]). *XPath* over labels Σ has constraints (ranged over by F, G, H), paths (ranged over by P, Q, R), and axes (ranged over by π):

$$\begin{array}{lcl} F & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg F \mid F \wedge F' \mid F \vee F' \mid P \mid A \quad (A \in \Sigma) \\ P & ::= & \pi[F] \mid P/P' \\ \pi & ::= & \mathsf{left} \mid \mathsf{right} \mid \mathsf{up} \mid \mathsf{down} \mid \mathsf{left}^+ \mid \mathsf{right}^+ \mid \mathsf{up}^+ \mid \mathsf{down}^+ \end{array}$$

For convenience we use abbreviations of XPath's axes, e.g. down for child, down^+ for descendant, etc. The definition of when a node in an ordered tree satisfies a constraint F is standard. We say that a tree satisfies a constraint F when its root node does, and we write $\mathcal{S}(D, F) \subseteq \mathrm{Tags}(\Sigma)^*$ for the set containing $\mathrm{stream}(T)$ for any T validated by D and satisfying F. We write $|F|$ for the routine definition of the size of a constraint F, generated by $|F \vee G| = |F \wedge G| = |F| + |G|$ and $|\pi[F]| = |\neg F| = 1 + |F|$.

It is also easy to see that XPath constraints can be enforced with finite memory over nonrecursive DTDs:

PROPOSITION 1. *For any nonrecursive DTD D and XPath*

```
not(//head[response and not(challenge)])
```

$\neg\mathsf{down}^{+}::\mathsf{head}[\mathsf{down}::\mathsf{response}[\mathsf{true}] \wedge \neg\mathsf{down}::\mathsf{challenge}[\mathsf{true}]]$

Figure 2: Example constraint F_1

constraint F, the set $\mathcal{S}(D, F)$ is regular.

3. XML STREAM FIREWALLING

The *XML stream firewalling problem* takes as input a nonrecursive DTD D over Σ, an XPath constraint F over Σ, and a string $s \in \Sigma^*$, and determines whether $s \in \mathcal{S}(D, F)$.

A *streaming* solution to the problem is one which reads s in strict left-to-right order. A *compiled* solution works in two phases: the first phase takes D and F as input and generates an intermediate result $\mathcal{I}(D, F)$, and the second phase takes D, F, $\mathcal{I}(D, F)$ and s as input and determines whether $s \in \mathcal{S}(D, F)$.

We will discuss the following time and space complexity classes for compiled, streaming solutions:

- *compilation* complexity given by the complexity of the first phase to process D and F,
- *per-stream* complexity given by the complexity of the second phase to process s, and
- *per-character* complexity given by the per-stream complexity for s, divided by $|s|$.

Throughout this paper, we will give time and space complexity in terms of a Random Access Machine.

Unfortunately, the minimal deterministic automaton recognizing $\mathcal{S}(D, F)$ for general XPath constraints may have doubly exponential size in F, and hence the amount of runtime space used may be prohibitive. We will thus restrict to a collection of constraints that can be implemented with greater space efficiency.

The notion of a subconstraint of a constraint is as usual. A top-level subconstraint of a constraint F is one which does not occur inside a subquery of F of the form $\pi[G]$. An XPath constraint has:

- an *explicit* rightward move if it contains a subconstraint $\mathsf{right}[F]$ or $\mathsf{right}^{+}[F]$,
- an *implicit* rightward move if it contains a subconstraint $\mathsf{up}[F]$ or $\mathsf{up}^{+}[F]$ where F has a top-level subconstraint $\mathsf{down}[G]$ or $\mathsf{down}^{+}[G]$,
- *supervised* leftward moves if every subconstraint $\mathsf{left}[F]$ or $\mathsf{left}^{+}[F]$ has F of the form $G \wedge \mathsf{up}[A]$, and
- *named* moves if every subconstraint $\pi[F]$ is of the form $\pi[A \wedge G]$ (and we write $\pi::A[G]$ for such named moves).

A constraint is *efficiently streamable* if it has no implicit or explicit rightward moves, every leftward move is supervised, and every move is named. For example, the "every response must have a challenge" example in the introduction is given in Figure 2, and is efficiently streamable.

We restrict our attention to efficiently streamable constraints, as i) the lack of rightward moves allows us to generate streaming solutions which process the document stream in left-to-right order, and ii) the supervision of leftward moves and the naming of vertical moves allows us to process such moves without needing to store a stack of potential target nodes. These conditions allow us (in Theorem 3) to solve the XML stream firewalling problem for constraint F in $O(|F|)$ per-stream space, compared to $O(|F| \times |\Sigma|)$ in [2].

Since $\mathcal{S}(D, F)$ is regular, it is tempting to choose the intermediate representation to be automata. Such a solution would entail constructing an automaton $\mathcal{A}(D, F)$ which *validates* F in conjunction with D (that is, where we have $\mathcal{L}(\mathcal{A}(D, F)) = \mathcal{S}(D, F)$). Unfortunately, even for efficiently streamable XPath, automata may require exponential space. There are two sources of this blowup: *sharing* of element definitions in DTDs, and the *product* construction used in building an automaton for conjunction or disjunction on XPath. Since the space usage of explicit automata is impractical, we will search for symbolic representations of automata which do not suffer from exponential blowup.

A solution to the XML stream firewalling problem is *fast-fail* when, given a string s such that there is no t such that $st \in \mathcal{S}(D, F)$, then any string su is rejected without reading u. For example, a fast-fail solution for the constraint in Figure 2 would reject any string beginning:

$$\langle\mathsf{messages}\rangle\langle\mathsf{message}\rangle\langle\mathsf{head}\rangle\langle\mathsf{response}/\rangle\langle/\mathsf{head}\rangle$$

Ideally, we would find a fast-fail solution for efficiently streamable XPath constraints with compilation time polynomial in D and F, per-stream space complexity and per-character time complexity polynomial in D and F. Unfortunately, such a fast-fail solution to the XML stream firewalling is likely to be impossible, as we can show:

THEOREM 1. *The following are equivalent:* **i)** *There is a fast-fail solution to the XML stream firewalling problem for efficiently streamable XPath constraints with compilation time polynomial in D and F and per-character time complexity polynomial in D and F)* and **ii)** *$P = PSPACE$*

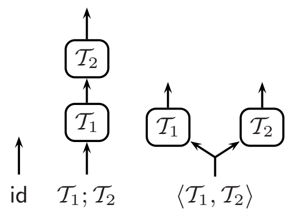
The proof that **i)** implies **ii)** works by reduction from the satisfiability problem for efficiently-streamable constraints, which can be shown to be PSPACE-hard. The other direction will utilize the transducer network construction given in the next section.

Since full fast-fail solutions are not feasible, we will first show that non-fast-fail solutions can be found with compilation time polynomial in D and F, and per-stream space and per-character time complexity linear in D and F; our solutions are based on transducer networks (TNs), and are the topic of Section 4. We will then seek heuristics for fast-fail solutions: our solutions compile TNs into binary decision diagrams (BDDs), and are discussed in Section 5.

4. TRANSDUCER NETWORKS

We first investigate a solution to the firewalling problem using transducer networks. We will use *finite state transducer networks*, as opposed to the pushdown transducer networks of [17, 16]. For nonrecursive DTDs, our prior work [2] shows that one can build small transducer networks that are sufficient for constraint processing. Here we will give a more detailed (and efficient) construction, which will be the basis of our work on fast-fail solutions in the next section.

A *synchronous transducer* $\mathcal{T}$ over input alphabet Σ and output alphabet Δ is an automaton over alphabet $\Sigma \times \Delta$.



$\mathcal{T}_2$
$\mathcal{T}_1$
$\mathcal{T}_1$
$\mathcal{T}_2$
id
$\mathcal{T}_1;\mathcal{T}_2$
$\langle\mathcal{T}_1,\mathcal{T}_2\rangle$

Figure 3: Graphical view of transducer networks

Write $q \xrightarrow{a/b} q'$ for $(q,(a,b),q') \in \rightarrow$. For $s \in \Sigma^*$ and $t \in \Delta^*$, write $q \overset{s/t}{\Longrightarrow} q'$ for the transitive reflexive closure of $\rightarrow$, i.e.:

$$q \overset{s/t}{\Longrightarrow} q' \text{ whenever } q = q_0 \xrightarrow{a_1/b_1} \cdots \xrightarrow{a_n/b_n} q_n = q' \\ \text{and } s = a_1 \dots a_n \text{ and } t = b_1 \dots b_n$$

The relation induced by a transducer $\mathcal{T}$ is defined to be $\mathcal{R}(\mathcal{T}) = \{(s,t) \mid I \ni q \overset{s/t}{\Longrightarrow} q' \in F\}$. A transducer is *sequential* whenever the transition relation is a partial function $\rightarrow : Q \times \Sigma \rightarrow \Delta \times Q$, and there is a single initial state. Note that automata over Σ are isomorphic with transducers over input alphabet Σ and output alphabet of size 1.

We will consider three operations on transducers: *identity*, *composition* and *product*, which are shown graphically in Figure 3. Let id be the *identity* transducer with input and output alphabet Σ (Σ will be clear from context, hence omitted in the notation). This is a one-state transducer with transitions:

$$0 \xrightarrow{A/A} 0 \qquad (A \in \Sigma)$$

which induces the identity relation on Σ^*:

$$\mathcal{R}(\mathsf{id}) = \{(s,s) \mid s \in \Sigma^*\}$$

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be transducers where each $\mathcal{T}_i$ has states Q_i, input alphabet Σ_i and output alphabet Δ_i. Composition $\mathcal{T}_1;\mathcal{T}_2$ takes the output of $\mathcal{T}_1$ and feeds it as input to $\mathcal{T}_2$. Formally, $\mathcal{T}_1;\mathcal{T}_2$ is defined when $\Delta_1 = \Sigma_2$, has input alphabet Σ_1 and output alphabet Δ_2, states $Q_1 \times Q_2$ and transitions:

$$(q_1,q_2) \xrightarrow{A/C} (q_1',q_2') \text{ where } q_1 \xrightarrow{A/B} q_1' \text{ and } q_2 \xrightarrow{B/C} q_2'$$

Composition induces the relation:

$$\mathcal{R}(\mathcal{T}_1;\mathcal{T}_2) = \{(s,u) \mid (s,t) \in \mathcal{R}(\mathcal{T}_1), (t,u) \in \mathcal{R}(\mathcal{T}_2)\}$$

Product $\langle\mathcal{T}_1,\mathcal{T}_2\rangle$ takes a copy of its input and feeds it to both $\mathcal{T}_1$ and $\mathcal{T}_2$. Formally, $\langle\mathcal{T}_1,\mathcal{T}_2\rangle$ is defined when $\Sigma_1 = \Sigma_2$, has input alphabet Σ_1 and output alphabet $\Delta_1 \times \Delta_2$, states $Q_1 \times Q_2$ and transitions:

$$(q_1,q_2) \xrightarrow{A/(B_1,B_2)} (q_1',q_2') \text{ where } q_1 \xrightarrow{A/B_1} q_1' \text{ and } q_2 \xrightarrow{A/B_2} q_2'$$

Product induces the relation:

$$\mathcal{R}(\langle\mathcal{T},\mathcal{T}'\rangle) = \{(s,t) \mid (s,\pi_1^*(t)) \in \mathcal{R}(\mathcal{T}_1), (s,\pi_2^*(t)) \in \mathcal{R}(\mathcal{T}_2)\}$$

Note that id is sequential, and if $\mathcal{T}_1$ and $\mathcal{T}_2$ are sequential, then so are $\mathcal{T}_1;\mathcal{T}_2$ and $\langle\mathcal{T}_1,\mathcal{T}_2\rangle$.

In this paper, we will use a textual representation of transducer networks, as it makes inductive definitions simpler. As Figure 3 shows, we can give a graphical reading to this textual form; examples are in Figures 4 and 6 which are discussed below. A *synchronous transducer network* with generators $\{\mathcal{T}_1,\dots,\mathcal{T}_n\}$ is a synchronous transducer definable by:

$$\mathcal{N} ::= \mathcal{T}_i \mid \mathsf{id} \mid \mathcal{N};\mathcal{N} \mid \langle\mathcal{N},\mathcal{N}\rangle$$

The size of a transducer network $|\mathcal{N}|$ extends the definition of $|\mathcal{T}|$ by $|\mathsf{id}| = 1$ and $|\mathcal{N}_1;\mathcal{N}_2| = |\langle\mathcal{N}_1,\mathcal{N}_2\rangle| = |\mathcal{N}_1| + |\mathcal{N}_2|$.

We can now show:

Theorem 2. *For any nonrecursive DTD D over Σ and efficiently streamable XPath constraint F over Σ, we can construct sequential transducer networks $\mathcal{N}(D)$ and $\mathcal{N}(F)$, and so construct automaton:*

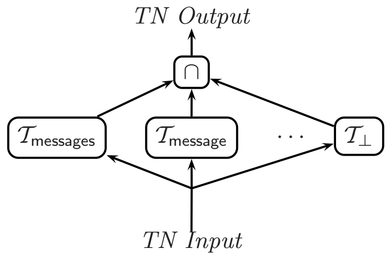
$$\mathcal{A}(D,F) = \mathcal{N}(D);\mathcal{N}(F);\mathsf{trueAtLast}$$

where trueAtLast *is the automaton for* $((\mathsf{true}+\mathsf{false})^*\,\mathsf{true})$, *such that:*

- *$\mathcal{A}(D,F)$ validates F in conjunction with D,*
- *$\mathcal{N}(D)$ is computed in time and space $O(|D||\Sigma|)$, and*
- *$\mathcal{N}(F)$ is computed in time and space $O(|F||\Sigma|)$.*

The network for a DTD contains generators $\mathcal{T}_A$ for each A in the DTD. It does not just validate the DTD, but also outputs the *parent tags* of the stream: when the generator $\mathcal{T}_A$ reads a $\langle B\rangle$ or $\langle/B\rangle$ child from an A node, it outputs A (on other nodes, it leaves the parent tag unconstrained). We include a transducer $\mathcal{T}_\perp$ to handle the special case of parent-tagging the root node with a $\perp$ tag. These parent tags will be useful to the constraint transducers.

The transducer $\mathcal{T}_A$ is based on the automaton for the language $D(A)$, with extra transitions to code parent tagging.



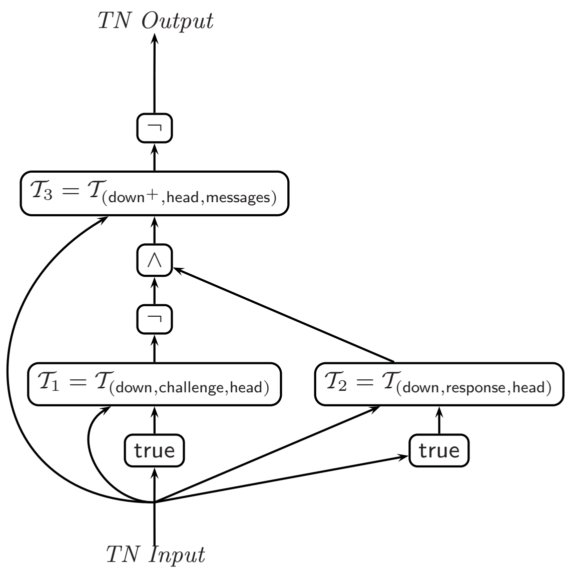
TN Output
$\cap$
$\mathcal{T}_{\mathsf{messages}}$
$\mathcal{T}_{\mathsf{message}}$
$\cdots$
$\mathcal{T}_\perp$
TN Input

$\langle\mathcal{T}_{\mathsf{messages}},\mathcal{T}_{\mathsf{message}},\dots,\mathcal{T}_\perp\rangle;\cap$

Figure 4: Transducer network built from D_1

TN	*Intermediate Output*			*TN*	*Final*
Input	$\mathcal{T}_\perp$	$\mathcal{T}_{\mathsf{messages}}$	$\mathcal{T}_{\mathsf{message}}$	*Output*	*State?*
⟨messages⟩	$\perp$	$*$	$*$	$\perp$	×
⟨message⟩	$*$	messages	$*$	messages	×
⟨head⟩	$*$	$*$	message	message	×
⟨response⟩	$*$	$*$	$*$	head	×
⟨/response⟩	$*$	$*$	$*$	head	×
⟨/head⟩	$*$	$*$	message	message	×
⟨/message⟩	$*$	messages	$*$	messages	×
⟨/messages⟩	$\perp$	$*$	$*$	$\perp$	✓

Figure 5: Example run of TN built from D_1


TN Output
¬
$\mathcal{T}_3 = \mathcal{T}_{(\mathsf{down}+,\mathsf{head},\mathsf{messages})}$
∧
¬
$\mathcal{T}_1 = \mathcal{T}_{(\mathsf{down},\mathsf{challenge},\mathsf{head})}$
$\mathcal{T}_2 = \mathcal{T}_{(\mathsf{down},\mathsf{response},\mathsf{head})}$
true
true
TN Input

$\langle \mathsf{id}, \langle\langle \mathsf{id}, \mathsf{true}\rangle; \mathcal{T}_1; \neg, \langle \mathsf{id}, \mathsf{true}\rangle; \mathcal{T}_2\rangle; \wedge\rangle; \mathcal{T}_3; \neg$

Figure 6: Transducer network built from F_1

TN Input		*Intermediate Output*			*TN*
Tag	*Parent*	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	*Output*
⟨messages⟩	⊥	×	×	×	✓
⟨message⟩	messages	×	×	×	✓
⟨head⟩	message	×	×	×	✓
⟨response⟩	head	×	×	×	✓
⟨/response⟩	head	×	×	×	✓
⟨/head⟩	message	×	✓	×	✓
⟨/message⟩	messages	×	×	×	✓
⟨/messages⟩	⊥	×	×	✓	×

Figure 7: Example run of TN built from F_1

The fact that the DTD is nonrecursive is used crucially here: transducer networks cannot recognize irregular languages such as those for recursive DTDs.

The network is built as the intersection of all of the generators, making use of an intersection transducer ∩ with input alphabet Σ^n and output alphabet Σ, inducing the intersection relation $\mathcal{R}(\cap) = \{((s, \dots, s), s) \mid s \in \Sigma^*\}$. It is routine to check that $\mathcal{R}(\langle \mathcal{N}_1, \dots, \mathcal{N}_n\rangle; \cap) = \mathcal{R}(\mathcal{N}_1) \cap \cdots \cap \mathcal{R}(\mathcal{N}_n)$.

We illustrate the construction for DTDs in Figure 4, which gives the topology of the network for the DTD D_1 from Figure 1, together with a sample run of the network in Figure 5.

The transducer network for a constraint in XPath is built recursively, and the structure of the network follows the structure of the formula. For each subconstraint of the form π::$A[F] \wedge B$, (that is, an axis π with target node labeled A satisfying F, and source node labeled B), we construct a generator $\mathcal{T}_{(\pi,A,B)}$. This transducer takes as input the parent-tagged input stream, together with a boolean stream giving the result of evaluating F. It produces as output a boolean stream giving the result of evaluating π::$A[F] \wedge B$. The logical operations have simple boolean generators.

The transducer networks for XPath constraints come in two flavors: *end-tagging* networks which produce output true when they reach the end tag of a selected node, or dual *begin-tagging* networks. End-tagging networks are used to handle the case of constraints without upward axes, and begin-tagging networks are used to handle the case of constraints without downward axes. In the case of efficiently streamable XPath constraints, we can nest these networks to combine upward and downward axes. For simplicity, we will discuss only end-tagging networks here, although both are used in the construction of the transducer network.

To construct an automaton from $\mathcal{T}(D)$ and $\mathcal{T}(F)$, we compose the network with a final automaton trueAtLast, which accepts boolean strings ending with true. Since the last symbol accepted by $\mathcal{T}(D)$ is guaranteed to be the end-tag for the root node, this automaton accepts strings where the root node satisfies F, and so accepts $\mathcal{S}(D, F)$.

We illustrate the construction for XPath in Figure 6, which gives the topology of the network for the constraint F_1 from Figure 2, together with a sample run of the network. Note that in the sample run, there is no string which is accepted by both the DTD and the constraint, and so the string is rejected (as it should be).

THEOREM 3. *Transducer networks provide a solution to the XML stream firewalling problem for efficiently streamable XPath, with compilation time $O((|D| + |F|)|\Sigma|)$, and with per-stream space and per-character time $O(|D| + |F|)$.*

However, the transducer network construction does not provide us with a fast-fail solution – this would require pruning sink states from the network. Pruning sink states is not a local operation on networks: there may be sink states of the network as a whole which are not sink states of any of the components. It is not clear how a pruning operation could even be expressed at the level of a rewriting of a transducer network, without first flattening the network to a single transducer. Sink states could be removed "on the fly", by checking reachability every time a state is reached (indeed, the checking can be done in PSPACE, which yields the second half of Theorem 1), but this approach would use exponential time. We will thus look for a formalism that can express both kinds of constraints and which supports compile-time heuristics for pruning sink states.

5. BINARY DECISION DIAGRAMS

We will now look at a different representation, using (*Reduced Ordered*) *Binary Decision Diagrams* [3] (*BDDs*), a compact representation of propositional logic formulae. In this work, we use BDDs as a compression technique for automata: BDDs will often give polynomial-space representations where an explicit automaton would be exponential-space. In this section, we show that for a class of transducer networks (which includes the TNs generated from DTDs and XPath), the BDD representation is polynomial in the size of the TN. Combined with prior results, this gives us polyspace representations of automata for DTDs and XPath.

Our motivation for introducing BDDs is to generate fast-fail XML stream firewalls. We know from Theorem 1 that we cannot do this in polynomial time (unless P = PSPACE), so we are interested in heuristic solutions. BDDs provide such a heuristic, as is demonstrated by the results in this section, and the experimental results in Section 7.

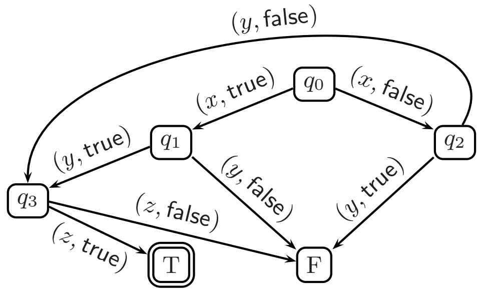


Figure 8: BDD for $(x \leftrightarrow y) \wedge z$

BDDs represent propositional logic formulas over atomic variables $\mathcal{V}$ as automata with an alphabet $\mathcal{V} \times \mathsf{Bool}$, as seen in Figure 8. Such an automaton gives possible assignments to variables, for example the above BDD includes the assignment $x = \mathsf{false}, y = \mathsf{false}, z = \mathsf{true}$. It is routine to check that the above BDD is a representation of the truth table for $(x \leftrightarrow y) \wedge z$. Moreover, it is a *canonical* representation (once we fix the variable order $x < y < z$) in the sense that it is the only BDD with that truth table.

Formally, a (reduced ordered) *binary decision diagram* (*BDD*) $\mathcal{B}$ over linearly ordered variables $(\mathcal{V}, \leq)$ is a deterministic automaton over alphabet $\mathcal{V} \times \mathsf{Bool}$ with no unreachable states, where every state is either:

- a chosen non-final state F with no outgoing transitions,
- a chosen final state T with no outgoing transitions, or
- a non-final state q with only two transitions of the form $q \xrightarrow{(x,\mathsf{true})} q'$ and $q \xrightarrow{(x,\mathsf{false})} q''$,

which is *ordered*:

- if $q \xrightarrow{(x,b)} q' \xrightarrow{(y,c)} q''$ then $x < y$,

and *reduced*:

- if $q \xrightarrow{(x,\mathsf{true})} \cdot \xleftarrow{(x,\mathsf{true})} q'$ and $q \xrightarrow{(x,\mathsf{false})} \cdot \xleftarrow{(x,\mathsf{false})} q'$ then $q = q'$, and
- if $q \xrightarrow{(x,\mathsf{true})} q'$ and $q \xrightarrow{(x,\mathsf{false})} q''$ then $q' \neq q''$.

A function $\sigma : \mathcal{V} \to \mathsf{Bool}$ is a *satisfying assignment* for $\mathcal{B}$ (written $\sigma \vDash \mathcal{B}$) when there is a string $s \in \mathcal{L}(\mathcal{B})$ such that $\sigma(x) = b$ for every $(x, b) \in s$. BDDs have the pleasant property that whenever two BDDs over $(\mathcal{V}, \leq)$ have the same satisfying assignments, then they are isomorphic. In particular, the only tautology is the BDD with initial state T, and the only unsatisfiable BDD is the one with initial state F.

Propositional logic operations can be performed directly at the BDD level. That is given BDDs $\mathcal{B}$ and $\mathcal{B}'$ over $(\mathcal{V}, \leq)$ we can construct the following BDDs, also over $(\mathcal{V}, \leq)$:

- true such that $\sigma \vDash \mathsf{true}$ for any σ,
- x such that $\sigma \vDash x$ whenever $\sigma(x) = \mathsf{true}$,
- $\mathcal{B} \wedge \mathcal{B}'$ such that $\sigma \vDash \mathcal{B} \wedge \mathcal{B}'$ whenever $\sigma \vDash \mathcal{B}$ and $\sigma \vDash \mathcal{B}'$,
- $\neg\mathcal{B}$ such that $\sigma \vDash \neg\mathcal{B}$ whenever $\sigma \nvDash \mathcal{B}$,
- $\exists x \,.\, \mathcal{B}$ such that $\sigma \vDash \exists x \,.\, \mathcal{B}$ whenever $\sigma, x \mapsto b \vDash \mathcal{B}$ for some b,
- $\mathcal{B}[x := y]$ such that $\sigma \vDash \mathcal{B}[x := y]$ whenever $\sigma, x \mapsto \sigma(y) \vDash \mathcal{B}$, and
- $\mathcal{B}[x := b]$ such that $\sigma \vDash \mathcal{B}[x := b]$ whenever $\sigma, x \mapsto b \vDash \mathcal{B}$.

Note that all of the above functions depend upon the variable ordering (which will always be clear from context, hence we omit it in the notation). Since BDDs are canonical, there is at most one BDD up to isomorphism satisfying each of the above properties, for a given ordering.

We will be interested in using BDDs to represent automata. This is relatively straightforward, once we fix a binary representation for the state set and alphabet. If we can represent the state set in Bool^k and the alphabet in Bool^j then we fix variables $\vec{x} \in \mathsf{Bool}^k$, $\vec{\imath} \in \mathsf{Bool}^j$ and $\vec{x}' \in \mathsf{Bool}^k$ then build propositions over those variables, for example trans is over $\vec{x}$ (the source state), $\vec{\imath}$ (the input action) and $\vec{x}'$ (the target state).

A *BDD representation* for an automaton with states $Q \subseteq \mathsf{Bool}^k$ over alphabet $\Sigma \subseteq 2^j$ is given by:

- a variable set $(\mathcal{V}, \leq)$,
- state variables $\vec{x}, \vec{x}' \in \mathcal{V}^k$ and action variables $\vec{\imath} \in \mathcal{V}^j$,
- BDDs state, init and final over variables $\{\vec{x}\}$, and
- BDD trans over variables $\{\vec{x}, \vec{\imath}, \vec{x}'\}$.

which represents an automaton with:

$$\begin{aligned} Q &= \{\vec{b} \mid \vec{x} \mapsto \vec{b} \vDash \mathsf{state}\} \\ I &= \{\vec{b} \mid \vec{x} \mapsto \vec{b} \vDash \mathsf{init}\} \\ F &= \{\vec{b} \mid \vec{x} \mapsto \vec{b} \vDash \mathsf{final}\} \\ \to &= \{(\vec{b}, \vec{c}, \vec{d}) \mid \vec{x} \mapsto \vec{b}, \vec{\imath} \mapsto \vec{c}, \vec{x}' \mapsto \vec{d} \vDash \mathsf{trans}\} \end{aligned}$$

Such BDD representations may be extremely efficient in space: there are families of automata of size $O(2^n)$ with BDD representations of size $O(n)$. However, they are not necessarily efficient in time, in that given q and a, computing the q' such that $q \xrightarrow{a} q'$ may require backtracking through the BDD. The source of this backtracking is when the value for a variable x'_i may depend on the value of x_j where $x'_i < x_j$: we may have to search both branches on x'_i to find a successful assignment.

We will say that a BDD over variables $\vec{x} < x < \vec{y}$ *uniquely determines* x whenever $\sigma(\vec{x}) = \sigma'(\vec{x})$ implies $\sigma(x) = \sigma'(x)$ for all satisfying assignments σ and σ'.

The BDD representation of an automaton is *runtime-friendly* whenever the variables $\vec{x}'$ are uniquely determined in trans. Such BDD representations can be executed in time linear in the bitlength of the state space, and so are good candidates for a runtime representation of an automaton:

Proposition 2. *Given a runtime-friendly BDD representation of an automaton $\mathcal{A}$ with states $Q \subseteq \mathsf{Bool}^k$ over alphabet $\Sigma \subseteq 2^j$, we can recognize $s \in \mathcal{L}(\mathcal{A})$ in $O(j+k)$ space and per-character time.*

For deterministic automata, we have efficient runtime-friendly BDD representations.

Proposition 3. *Any deterministic automaton $\mathcal{A}$ with states $Q \subseteq \mathsf{Bool}^k$ over alphabet $\Sigma \subseteq \mathsf{Bool}^j$ has a runtime-friendly BDD representation computed in time and space $O((j + k)|Q||\Sigma|)$.*

The BDD representation of a transducer is given in the same way as an automaton, but the action variables are partitioned into $\vec{\imath}$ for input and $\vec{o}$ for output. We can improve on Proposition 3 in the case of sequential transducers:

Proposition 4. *Any sequential transducer $\mathcal{T}$ with states $Q \subseteq \mathsf{Bool}^k$ over input alphabet $\Sigma \subseteq \mathsf{Bool}^i$ and output alphabet $\Delta \subseteq \mathsf{Bool}^j$ has a runtime-friendly BDD representation computed in time and space $O((i+j+k)|Q||\Sigma|)$.*

This can be generalized to a network of transducers. If $\mathcal{N}_1$ and $\mathcal{N}_2$ have BDD representations $\mathcal{B}_1$ and $\mathcal{B}_2$ respectively, where each $\mathcal{B}_i$ has state variables $\vec{x}_i$ and $\vec{x}'_i$, input variables $\vec{\imath}_i$ and output variables $\vec{o}_i$, then $\langle \mathcal{N}_1, \mathcal{N}_2 \rangle$ has BDD representation:

$$\begin{array}{rclrcl} \vec{x} & = & \vec{x}_1, \vec{x}_2 & \mathsf{state} & = & \mathsf{state}_1 \wedge \mathsf{state}_2 \\ \vec{\imath} & = & \vec{\imath}_1 & \mathsf{init} & = & \mathsf{init}_1 \wedge \mathsf{init}_2 \\ \vec{o} & = & \vec{o}_1, \vec{o}_2 & \mathsf{final} & = & \mathsf{final}_1 \wedge \mathsf{final}_2 \\ \vec{x}' & = & \vec{x}'_1, \vec{x}'_2 & \mathsf{trans} & = & \mathsf{trans}_1 \wedge \mathsf{trans}_2[\vec{\imath}_2 := \vec{\imath}_1] \end{array}$$

Similarly, $\mathcal{N}_1;\mathcal{N}_2$ has BDD representation:

$$\begin{array}{rclrcl} \vec{x} & = & \vec{x}_1, \vec{x}_2 & \mathsf{state} & = & \mathsf{state}_1 \wedge \mathsf{state}_2 \\ \vec{\imath} & = & \vec{\imath}_1 & \mathsf{init} & = & \mathsf{init}_1 \wedge \mathsf{init}_2 \\ \vec{o} & = & \vec{o}_2 & \mathsf{final} & = & \mathsf{final}_1 \wedge \mathsf{final}_2 \\ \vec{x}' & = & \vec{x}'_1, \vec{x}'_2 & \mathsf{trans} & = & \exists \vec{o}_1 \,.\, (\mathsf{trans}_1 \wedge \mathsf{trans}_2[\vec{\imath}_2 := \vec{o}_1]) \end{array}$$

This construction leaves open an issue that may have an exponential impact on the size of the final BDD: the ordering on variables.

We first consider the kinds of variable orderings needed to perform the composition $\mathcal{N}_1;\mathcal{N}_2$ of transducers efficiently. This includes two potential sources of inefficiency:

- We need to form a conjunction, which we do by an appropriate "gluing" of the BDD $\mathsf{trans}_2[\vec{\imath}_2 := \vec{o}_1]$ to the leaves of trans_1. This gluing can be done without blowup if the variable ordering on $\mathcal{B}_1$ places the assignment to variables $\vec{o}_1$ at the leaves of trans_1, and the decisions based on variables $\vec{\imath}_2$ at the root of trans_2. This can be achieved by requiring $\vec{o}_1$ to be maximal in the variable order used by trans_1 and $\vec{\imath}_2$ to be minimal in the variable order used by trans_2.

- We need to existentially quantify over the output variables $\vec{o}_1$ of $\mathcal{B}_1$, which we do by an appropriate "deletion" of nodes in the BDD. This deletion in a general BDD is not a cheap operation, as it may require merging of the child nodes, but it is cheap on uniquely determined variables. As long as we require (as above) the variables $\vec{o}_1$ to be maximal in the variable order for trans_1, they are guaranteed to be uniquely determined.

For this reason, we define a BDD representation of a transducer to be *composition-friendly* whenever the variable order has the input variables $\vec{\imath}$ minimal and the output variables $\vec{o}$ maximal. The variable order we choose for $\mathcal{N}_1;\mathcal{N}_2$ is:

$$\vec{\imath}_1 < (\vec{x}_1, \vec{x}'_1) < (\vec{x}_2, \vec{x}'_2) < \vec{o}_2$$

which gives us a linear-space construction for composition:

Proposition 5. *If $\mathcal{N}_1$ and $\mathcal{N}_2$ are sequential transducers with runtime- and composition-friendly representations $\mathcal{B}_1$ and $\mathcal{B}_2$, then we can find a runtime- and composition-friendly representation for $\mathcal{N}_1;\mathcal{N}_2$ of size $|\mathcal{B}_1| + |\mathcal{B}_2|$.*

Unfortunately, the same argument does not hold for the product operation $\langle \mathcal{N}_1, \mathcal{N}_2 \rangle$, and the best result we can get creates a blowup in the size of the alphabet:

Proposition 6. *If $\mathcal{N}_1$ and $\mathcal{N}_2$ are transducer networks with input alphabet Σ and output alphabets Δ_1 and Δ_2, and with runtime- and composition-friendly representations $\mathcal{B}_1$ and $\mathcal{B}_2$, then we can find a runtime- and composition-friendly representation for $\langle \mathcal{N}_1, \mathcal{N}_2 \rangle$ of size $|\Sigma||\mathcal{B}_1| + |\Delta_1||\mathcal{B}_2|$.*

This multiplicative factor for each use of product results in a potential exponential blowup in the size of a BDD for a general transducer network, and so we will need to exploit the special properties of transducer networks which we generated for DTDs and XPath constraints.

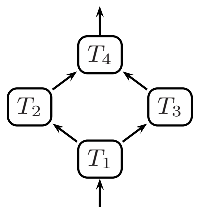
We first consider the transducer networks generated from DTDs. These are of a very particular form:

$$\mathcal{N} = \langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle; \cap$$

where each $\mathcal{T}_i$ is deterministic (but not necessarily sequential), and $\mathcal{N}$ as a whole is sequential: we will call transducer networks in this form *flat intersection* networks. A nice property of flat intersection networks is that on any transition of the whole network, we have that *every* generator $\mathcal{T}_i$ has the same input and output. For this reason, we can show that they have small representations:

Proposition 7. *Any flat intersection network $\mathcal{N}$ with states $Q \subseteq \mathsf{Bool}^k$ over input alphabet $\Sigma \subseteq \mathsf{Bool}^i$ and output alphabet $\Delta \subseteq \mathsf{Bool}^j$ has a runtime- and composition-friendly BDD representation in time and space $O((i+j+k)|\mathcal{N}||\Sigma||\Delta|)$.*

The case of the transducer networks generated from XPath constraints is slightly more complex. As the example in Figure 6 shows, these networks are very tree-like: the only sharing is of the top-level input to the network. In particular, XPath constraints never generate networks containing "diamonds" such as:



Formally, we define a transducer network to be *boolean* whenever each generator has output alphabet Bool, and *tree-like* when it is can be generated by the following grammar:

$$\mathcal{N} ::= \mathsf{id} \mid \langle \mathcal{N}, \mathcal{N} \rangle; \mathcal{T}_i$$

The reader can verify that the above grammar prevents "diamonds". We shall now consider the size of the BDD representation of a boolean tree-like transducer network. We shall first consider the case when $|\Sigma| = 1$. The general case follows from this, since we can decompose the transition BDD as a disjunction over all possible inputs. The interesting case in computing a bound on the size of a boolean tree-like network is $\mathcal{N} = \langle \mathcal{N}_1, \mathcal{N}_2 \rangle; \mathcal{T}_3$ where the components have BDD representations $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$ respectively. An application of Propositions 5 and 6 gives us the bound $|\mathcal{B}_1| + 2|\mathcal{B}_2| + |\mathcal{B}_3|$. However, we can swap the order of $\mathcal{N}_1$ and $\mathcal{N}_2$, in which case the same propositions gives us bound $|\mathcal{B}_2| + 2|\mathcal{B}_1| + |\mathcal{B}_3|$.

Unfortunately, this still gives a multiplicative factor for each use of product, which raises the potential of exponential blowup. Fortunately, we can choose to double the smaller of $\mathcal{B}_1$ and $\mathcal{B}_2$, and so the size of the BDD representation of a boolean tree-like network is bounded by the function $f : \mathbb{N} \to \mathbb{N}$ defined:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= \max\{f(i) + 2f(j) \mid n > i \geq j \in \mathbb{N}, i + j = n\} \end{aligned}$$

Now consider the function $g : \mathbb{R} \to \mathbb{R}$ defined:

$$g(x) = x^{\log_2(3)}$$

It is routine to verify that:

$$\max\{p^{\log_2(3)} + 2(1-p)^{\log_2(3)} \mid 0.5 \leq p < 1\} = 1$$

and hence that, for any $n \in \mathbb{N}$:

$$\begin{aligned} g(n) &= n^{\log_2(3)} \max\{p^{\log_2(3)} + m(1-p)^{\log_2(3)} \mid 0.5 \leq p < 1\} \\ &= \max\{(pn)^{\log_2(3)} + ((1-p)n)^{\log_2(3)} \mid 0.5 \leq p < 1\} \\ &= \max\{g(pn) + g((1-p)n) \mid 0.5 \leq p < 1\} \\ &\geq \max\{g(i) + g(j) \mid n > i \geq j \in \mathbb{N}, i + j = n\} \end{aligned}$$

Thus, since g dominates the defining equation for f, we have that $f(n) \leq n^{\log_2(3)}$. We thus have the following bound:

PROPOSITION 8. *Any boolean tree-like network $\mathcal{N}$ with states $Q \subseteq \mathsf{Bool}^k$ over input alphabet $\Sigma \subseteq \mathsf{Bool}^j$ has a runtime- and composition-friendly BDD representation in time and space $O((j+k)|\mathcal{N}|^{\log_2(3)}|\Sigma|)$.*

We now have enough results to state the complexity of our BDD solution to the XML stream firewalling problem.

THEOREM 4. *BDDs provide a solution to the XML stream firewalling problem for efficiently streamable XPath over alphabet $\Sigma \subseteq \mathsf{Bool}^j$, with compilation time and space $O(j(|D|^2 + |F|^{\log_2(3)}) \times |\Sigma|^2)$, and with per-stream space and per-character time per-character time $O(|D| + |F|)$.*

The linear per-character time follows because the number of state variables in the BDDs is linear, and this bounds the depth of the BDD. The quadratic dependence on the alphabet Σ above is caused by the need for the XPath transducers to use parent-tagged input.

The advantage of BDDs for fast-fail firewalling is is that pruning of sink states can be performed in a natural manner, with an application of symbolic model-checking techniques [8]. We can approximate the reachability relation on states by BDDs reach_n over variables $\vec{x}, \vec{x}'$ defined:

$$\begin{aligned} \mathsf{reach}_1 &= (\vec{x} = \vec{x}') \vee (\exists \vec{\imath} \,.\, \exists \vec{o} \,.\, \mathsf{trans}) \\ \mathsf{reach}_{n+1} &= \exists \vec{x}'' \,.\, (\mathsf{reach}_n[\vec{x}' := \vec{x}''] \wedge \mathsf{reach}_n[\vec{x} := \vec{x}'']) \end{aligned}$$

This sequence will reach a fixed point reach after $\log_2(|s|)$ iterations, where s is a maximal acyclic path through the automaton; since BDDs are canonical, we can detect reaching a fixed point in constant time. We can then prune sink states by defining:

$$\mathsf{state}' = \exists \vec{x}' \,.\, (\mathsf{reach} \wedge \mathsf{final}[\vec{x} := \vec{x}'])$$

In the worst case, reach_{n+1} may be double the size of reach_n, and so state' may be exponentially larger than state; this

```
typedef unsigned int nfaE_t;
nfaE_t nfaE_u[473][16] = { ... }
int nfaE_v[1061][16] = { ... }
int nfaE_w[256] = { ... }
int nfaE_sink(nfaE_t q) { return (q == 0x1); }
int nfaE_accept(nfaE_t q) { return (q&0x1 == 0x1) && (q != 0x1); }
nfaE_t nfaE_init() { return 0x0; }
nfaE_t nfaE_next(nfaE_t q, char c) {
  int v0 = nfaE_w[c & 0xFF];
  int v1 = nfaE_v[v0][(q >> 24) & 0xF];
  int v2 = nfaE_v[v1][(q >> 20) & 0xF];
  int v3 = nfaE_v[v2][(q >> 16) & 0xF];
  int v4 = nfaE_v[v3][(q >> 12) & 0xF];
  int v5 = nfaE_v[v4][(q >> 8) & 0xF];
  int v6 = nfaE_v[v5][(q >> 4) & 0xF];
  nfaE_t u = nfaE_u[v6][q & 0xF];
  nfaE_t r = q ^ u;
  return (u == 0x1? 0x1: r);
}
```

Table 1: Table-based C code for XPathMark Q1

blowup is to be expected, due to Theorem 1. However, in many cases, BDDs give an acceptable heuristic for generating fast-fail firewalls, as will be demonstrated in the next section.

6. IMPLEMENTATION

Tree Fort is an implementation of the transducer network given in Theorem 2, using a BDD as in Section 5. The implementation contains two optimizations for common cases in automata: *near self-loops* and *transitions to sink states*:

- In the transition relation trans, it is common for most state variables to remain unchanged, that is $x_i = x_i'$ for most i. We call such cases *near self-loops*, and note that they are common in transducer networks since on most inputs, most of the transducers in the network stay in the same state. In the straightforward representation of the transition relation trans, near self-loops do not result in sharing, since x_i' depends on x_i. There are more opportunities for sharing in the BDD for the following *state offset representation*:

  $$\mathsf{trans}' = \exists \vec{x}' \,.\, ((\vec{x}' = (\vec{x} \,\mathsf{xor}\, \vec{x}'')) \wedge \mathsf{trans})$$

  In trans', when $x_i = x_i'$ we have that $x_i'' = \mathsf{false}$, which does not depend on x_i, so trans' often has a smaller BDD representation than trans.

- The state offset representation has a drawback for sink states. For every state that transitions into a sink state, there is a distinct offset, and hence there is no opportunity for sharing. To counter this problem we use an alternative representation for sink states. We find an assignment $\vec{b}$ such that:

  $$\vec{x}'' \mapsto \vec{b} \vDash \nexists \vec{x}, \vec{x}' \,.\, \mathsf{trans}'$$

  that is, $\vec{b}$ is a value never assigned by trans', so can be used as a "null" value. The BDD we implement is:

  $$\mathsf{trans}'' = \mathsf{trans}' \vee ((\vec{x}'' = \vec{b}) \wedge (\nexists \vec{x}'' \,.\, \mathsf{trans}'))$$

  We can then recover trans from trans'' as:

  $$\mathsf{trans} = \exists \vec{x}'' \,.\, ((\vec{x}'' \neq \vec{b}) \wedge (\vec{x}' = (\vec{x} \,\mathsf{xor}\, \vec{x}'')) \wedge (\mathsf{trans}''))$$

  Since the original automaton was deterministic, $\vec{x}''$ is uniquely determined in trans''.

- To execute a transition of the state machine given by the variable ordering described in Section 5, we would need to determine the output a few variables at a time (since input and output were interleaved). In Tree Fort the ordering we use for trans'' has $\vec{x} < \vec{x}''$, which allows for a simple trie-like implementation, since the assignment to $\vec{x}''$ is given only at leaf nodes. This can be seen in Table 1 (where both the "null offset" $\vec{b}$ and the chosen sink state are `0x1`). There is a potential blowup in this ordering, but in practice space was acceptable.

The XPathMark test suite is based on returning nodesets for queries, and not on firewalling; furthermore, many prior XPath processors that we wished to compare against implement only nodeset queries. We thus extended our approach to implement nodeset queries:

- As we did with constraints, we build an automaton to recognize the end-tag of any selected node. However, when it accepts, we search backwards through the document to find the matching begin-tag, which we copy to the output stream. This requires a buffer of size equal to the largest candidate node in the document.

- We implement nodeset queries by translating XPath paths P into XPath constraints $\mathcal{C}(P)$. The interesting cases for this translation are:

  $$\mathcal{C}(P/\pi[F]) = \pi^{-1}[\mathcal{C}(P)] \wedge F \qquad \mathcal{C}(\pi[F]) = \pi^{-1}[A_0] \wedge F$$

  for example:

  $$\mathcal{C}(\mathsf{down}{::}B/\mathsf{down}^{+}{::}C) = C \wedge \mathsf{up}^{+}{::}B[\mathsf{up}{::}A_0]$$

  Note that this translation inverts the axes, which impacts the definition of "efficiently streamable". A nodeset query is considered to be efficiently streamable when its translation is.

- Even when we are implementing nodeset queries, we can still benefit from a fast-fail implementation. An implementation of nodeset queries is fast-fail whenever, given an input string s such that for any t the output generated by s and st are equal, the implementation terminates without reading t. For example, given a DTD including $A \mapsto B^{*}C^{*}$, the nodeset query $\mathsf{down}{::}B$ can fast-fail after reading input $\langle A\rangle\langle B/\rangle^{n}\langle C\rangle$ and generating output $\langle B/\rangle^{n}$.

The implementation includes some additional features:

- We extend XPath with regex-matching on PCDATA, where regexes are given in Augmented Backus Normal Form (ABNF) syntax. The constraint $P = R$ is true whenever a node in the nodeset returned by P has PCDATA recognized by regex R.

- We construct a transducer network that parses as well as validates; that is, we generate automata recognizing subsets of char^{*} rather than $\mathrm{Tags}(\Sigma)^{*}$. The transducer for the DTD is therefore nondeterministic (as the regex $(\langle A\rangle \cdots) + (\langle B\rangle \cdots)$ both begin with the character $\langle$) but can be sequentialized.

- The proofs of the size bounds for BDDs generate fresh variables, which gives good complexity bounds, but at the cost of large state size. In the implementation, we reuse variables where possible.

$Q1$	`/site/regions/*/item`
$Q7$	`//keyword/ancestor-or-self::mail`
$Q9$	`/site/open_auctions/open_auction[@id='open_auction0']`
	`/bidder/preceding-sibling::bidder`
$Q36$	`/site/regions/*/item[contains(description,'gold')]`
$Q42$	`/site/regions/*/item`
	`[string-length(normalize-space(string(description))) > 1000]`

Table 2: Sample of Original XPathMark queries

$Q1$	`/site/regions//item`
$Q7$	`//keyword/ancestor-or-self::mail`
$Q9$	cannot be implemented as a DFA
$Q36$	`/site/regions//item[description = (*OCTET 'gold' *OCTET)]`
$Q42$	`/site/regions//item[description = 1000*OCTET]`

Table 3: Sample of Modified XPathMark queries

$Q1$	`//UniProtKB_ID`
$Q2$	`//UniProtKB[UniProtKB_ID/text() = '1433F_HUMAN']`
$Q3$	`//Protein_Name_and_ID/UniProtKB`
	`[UniProtKB_ID/text() = '1433F_HUMAN']`
	`/following-sibling::IPI`
$Q4$	`//GENERAL_INFORMATION/Complex`
	`[not(preceding-sibling::Tissue_Specificity)]`
$Q5$	`//Protein_Name[not(preceding::Protein_Name)]`

Table 4: iProClass queries

- The run-time representation of the automaton is BDD-based, but uses a 16-way decision rather than a 2-way decision. Increasing the fan-out lowers the number of memory accesses, at the cost of larger arrays. In a RAM model, this results in a space–time tradeoff, but in practice increasing the space usage has an impact on time, since L2 cache performance is impacted. Experimentally, we found that a 16-way branching produced optimal runtimes.

The implementation is approximately 13k lines of Standard ML [20] code, and generates a direct table-based C implementation of the 16-way decision diagram for trans'', as seen in Table 1. Two alternate implementation strategies were investigated:

- A *code-based* rather than table-based implementation was generated, which used switch-statements rather than array lookups to implement branching. This resulted in no change in space usage, but a 2–3$\times$ blowup in runtime (conjectured to be due to less accurate branch prediction by the CPU).

- *Sparse arrays* were used as an alternative implementation of missing transitions. (Sparse arrays are a data structure which efficiently represent "arrays with holes" with constant-time lookup.) This resulted in a 2–4$\times$ decrease in space usage, with a 20% increase in runtime (conjectured to be due to larger numbers of local variables, and so more variables being placed on the stack rather than in the few available registers on the x86 architecture used for testing).

7. EXPERIMENTAL RESULTS

For evaluation, we have used our implementation on the XPathMark query set [15] against a nonrecursive variant of

Implemented (26):	
No rewriting required (9)	{ $Q2, Q3, Q4, Q6, Q7, Q12, Q22, Q23, Q24$ }
Rewritable in input fragment (17)	{ $Q1, Q5, Q8, Q10, Q13, Q15, Q25, Q31, Q32, Q36, Q37, Q38, Q39, Q40, Q41, Q42, Q47$ }
Unimplemented (21):	
Is not end-determined (3)	{ $Q9, Q11, Q30$ }
Uses missing features (18)	{ $Q14, Q16, Q17, Q18, Q19, Q20, Q21, Q26, Q27, Q28, Q29, Q33, Q34, Q35, Q43, Q44, Q45, Q46$ }

Table 5: XPathMark query coverage

the XMark DTD, and on the Integrated Protein Knowledgebase for H. Sapiens [13].

The XPathMark [15] query set is based on the XMark [18] document set. The XMark auction DTD is recursive, due to the presence of text mark-up elements such as `<parlist>` and `<listitem>` which may be arbitrarily nested. The XPathMark synthetic reference document D1 was edited to remove approximately 30k `<parlist>` and `<listitem>` tags: the resulting D1-norec document is 115.9MB compared to the original 116.5MB. XPathMark contains 47 queries, some of which are given in Table 2.

Tree Fort does not support all of XPath, only the efficiently streamable fragment, without features such as namespaces and comment nodes, some of which are used by XPathMark. Tree Fort does not support functions or data joins, but does allow regular expression matching on PCDATA, for example $Q36$ (in Table 2) is rewritten to remove the wildcard `*` axis, and the `contains` function, to give the equivalent (under the XMark auction DTD) query (given in Table 3).

Efficiently streamable formulae are *end-determined* [2] in the sense that membership of a node in a nodeset can be determined when the end-tag is read. This is a requirement for the automaton implementation strategy, and so Tree Fort is unable to process non-end-determined queries such as $Q9$; indeed, *any* approach which uses buffer space bounded by the size of the largest `bidder` node will not be able to process this query.

We also tested our constraint-checking code on an additional dataset. The Integrated Protein Knowledgebase (iProClass) is a bioinformatics application, containing links to other biological databases. The iProClass DTD is nonrecursive, so no special treatment of nested documents is required. The H. Sapiens knowledgebase is 485.6MB, and was used for testing. The queries used for benchmarking are in Table 4. Note that these queries are all efficiently streamable, and that $Q5$ is fast-fail: once the first `Protein_Name` is returned, the query is unsatisfiable.

Experiments were carried out on a 1GB 2.8GHz P4 running Ubuntu Linux, gcc 4.0.3 and Sun Java 1.5.0. We compare our implementation to the XPath engines GCX [19], Spex [16], XMLtk [11], XSLTproc [21], XSQ [17] and YFilter [9]. We also compare our implementation to XMLWF [7], which only checks for a well-formed XML document.

The XPathMark test coverage results are given in Table 5. Comparing with Table 6 (where blank space denotes queries not covered), we see that Tree Fort has better test coverage results than any tool other than XSLTproc, which implements all of XPath (but is tree- rather than stream-based). In addition, Tree Fort is performing DTD validation (in this case, against the auction DTD), which is not the case of any other tool.

	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	$Q7$	$Q8$
Tree Fort	**1.62**	2.87	2.23	2.18	**1.52**	2.74	**1.29**	4.20
XMLtk	2.16	1.76	**2.08**	**1.74**		**2.04**	2.02	
GCX	3.38	**1.70**	2.28	2.21		2.79	2.38	
Spex	28.5	17.0	20.0			29.7	26.4	
XSQ	21.7	9.91	11.6					
XSLTproc	116	11.0	–	–	31.4	107	24.0	**3.62**
YFilter	–	–	–					

	$Q9$	$Q10$	$Q12$	$Q13$	$Q15$	$Q17$	$Q21$
Tree Fort		2.26	**1.38**	**4.17**	**2.56**		
XMLtk	**1.81**	**1.79**					1.77
GCX						6.60	
Spex	28.5	17.0	20.0			29.7	26.4
XSQ	21.7	9.91	11.6				
XSLTproc	3.52	114	6.08	–	35.7	8.41	4.28
YFilter							

	$Q22$	$Q23$	$Q24$	$Q25$	$Q31$	$Q32$	$Q36$
Tree Fort	**1.54**	**1.76**	**1.57**	**1.76**	**2.59**	**1.42**	**1.37**
XMLtk							
GCX		2.18	2.18				
Spex	23.0						
XSQ							
XSLTproc	31.5	22.7	38.9	3.48	6.22	118	4.81
YFilter							

	$Q37$	$Q38$	$Q39$	$Q40$	$Q41$	$Q42$	$Q47$
Tree Fort	**1.72**	**1.74**	**1.72**	**1.83**	**1.78**	**2.57**	**1.58**
XMLtk							
GCX							2.18
Spex	23.0						
XSQ							
XSLTproc	4.41	4.52	4.50	13.2	12.6	23.2	38.9
YFilter							

XMLwf	1.60	(– = Timeout after 120s)

Table 6: XPathMark query performance

The performance results for XPathMark and iProClass are shown in Tables 6 and 7. Note that Tree Fort is much faster than many of the implementations, such as YFilter and XSQ. Its performance is competitive with GCX and XMLtk: in all but one test Tree Fort is either fastest or second-fastest. Throughput is 200–400Mbps. Queries $Q1$ and $Q7$ (where Tree Fort outperforms all other processors) are both ones which can fast-fail (in both cases, the query only returns subtrees of `<regions>`, which is guaranteed by the DTD to be the first child of the root node).

The space usage is shown in Tables 8 and 9. We see that:

- The automata have 13k–7m states; the largest automaton by a factor of 60× is Q42, which includes the 1000-state regex `1000*OCTET`.
- The BDD-based representation for the automata gives between 5× (Q13) and 8000× (Q42) compression ratios compared to an explicit table-based representation.
- The fixed space usage in all cases is 109–891KB. The per-stream space usage is 4 or 8 bytes.

In summary, on these experiments we have achieved our goals of low per-input-character time (achieving throughput of 200–400Mbps), low per-stream space (4 or 8 bytes), and fast-fail execution.

	Q1	*Q2*	*Q3*	*Q4*	*Q5*
Tree Fort	**11.0**	**16.3**	**16.5**	**10.8**	**0.01**
XMLtk	12.1				
GCX	12.7			13	
Spex	106	–	87		
XSQ	55.7			55.2	56.4
XSLTproc	–	–	–	–	–
YFilter	–				
XMLwf	11.2	(– = Timeout after 120s)			

Table 7: iProClass query performance

	Q1	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	*Q7*
(N)	13k	19k	25k	21k	13k	30k	20k
(X)	13MB	19MB	25MB	21MB	13MB	30MB	20MB
(B)	116KB	273KB	227KB	238KB	134KB	270KB	153KB
(S)	4B	4B	4B	4B	4B	4B	4B
	Q8	*Q10*	*Q12*	*Q13*	*Q15*	*Q22*	*Q23*
(N)	21k	51k	25k	18k	18k	13k	18k
(X)	21MB	51MB	25MB	18MB	18MB	13MB	18MB
(B)	379KB	194KB	135KB	381KB	243KB	136KB	224KB
(S)	8B	8B	4B	8B	4B	4B	4B
	Q24	*Q25*	*Q31*	*Q32*	*Q36*	*Q37*	
(N)	16k	16k	28k	13k	120k	17k	
(X)	16MB	16MB	28MB	13MB	120MB	17MB	
(B)	215KB	183KB	219KB	247KB	109KB	288KB	
(S)	4B	4B	4B	4B	4B	4B	
	Q38	*Q39*	*Q40*	*Q41*	*Q42*	*Q47*	
(N)	16k	17k	17k	16k	7134k	16k	
(X)	16MB	17MB	17MB	16MB	7134MB	16MB	
(B)	196KB	219KB	238KB	212KB	891KB	188KB	
(S)	8B	8B	8B	8B	8B	4B	

(N) = no. of states in DFA (X) = explicit table-based DFA
(B) = BDD-represented DFA (S) = per-stream space usage

Table 8: XPathMark space usage

8. FUTURE WORK

In this paper we have defined, investigated, and proposed solutions for the XML stream firewalling problem. Many issues are are left open for future work:

- The input stream is required to be an XML document: can we apply the same techniques to other message formats? In particular, will the techniques scale to support message formats which do not support deterministic parsers?

- Can we find similar results for recursive DTDs? We cannot hope to validate documents in finite space. However, our techniques should allow us to get efficient symbolic representations of wider classes of automata – e.g. automata that can make use of the ancestor chain of a current element.

9. REPEATABILITY ASSESSMENT RESULT

Tables 6 and 7 have been verified by the SIGMOD repeatability committee.

10. REFERENCES

[1] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. VLDB*, 2000.

[2] M. Benedikt and A. Jeffrey. Efficient and expressive tree filters. In *Proc. FSTTCS*, 2007.

[3] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comp.*, 35(8):677–691, 1986.

[4] Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proc. ICSE*, 2001.

[5] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proc. VLDB*, 2002.

[6] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. ICDE*, 2000.

[7] J. Clark. Expat – The XML Parser Toolkit. `http://www.jclark.com/xml/expat.html`.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[9] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. VLDB*, 2004.

[10] M. Franceschet. XPathMark: an XPath benchmark for XMark generated data. In *Proc. XSYM*, 2005.

[11] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proc. ICDT*, 2003.

[12] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proc. SIGMOD*, 2003.

[13] The iProClass database. `http://pir.georgetown.edu`.

[14] M. Marx. Semantic Characterizations of Navigational XPath. *SIGMOD Record*, 34:41–46, 2005.

[15] M.Francechet. XPathMark: an XPath benchmark for XMark generated data. In *Proc. XSYM*, 2005.

[16] D. Olteanu. Streamed and Progressive Evaluation of XPath. *IEEE TKDE*, 19(7), July 2007.

[17] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *Proc. SIGMOD*, 2003.

[18] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[19] M. Schmidt, S. Scherzinger, and C. Koch. The GCX System: Dynamic Buffer Minimization in Streaming XQuery Evaluation. In *Proc. VLDB*, 2007.

[20] Standard ML. `http://www.standardml.org`.

[21] D. Veillard. The XML C Parser and Toolkit of Gnome: XSLT, 2003. `http://xmlsoft.org/XSLT`.

	Q1	*Q2*	*Q3*	*Q4*	*Q5*
(N)	22k	29k	29k	22k	22k
(X)	22MB	29MB	29MB	22MB	22MB
(B)	363KB	530KB	547KB	365KB	361KB
(S)	4B	4B	4B	4B	4B

(N) = no. of states in DFA (X) = explicit table-based DFA
(B) = BDD-represented DFA (S) = per-stream space usage

Table 9: iProClass space usage