A Trace Semantics for Positive Core XPath

Pieter H. Hartel
Dept. of Computer Science, Univ. of Twente, The Netherlands
Pieter.Hartel@utwente.nl

Abstract

We provide a novel trace semantics for positive core XPath that exposes all intermediate nodes visited by the query engine. This enables a detailed analysis of all information relevant to the query. We give two examples of such analyses in the form of access control policies. We translate positive core XPath into Linear Temporal Logic, showing that branching structures can be linearised effectively. We use the SPIN model checker in a proof of concept implementation to resolve the queries, and to perform access control. The performance of the implementation is competitive.

1. Introduction

Many approaches towards Access control on XML data use XPath (directly or indirectly) both for the queries and for access control (e.g. [5]). We are interested in combining a more flexible, logical approach [1] to access control, with the standard XPath based querying. An XPath (version 1.0 [8]) query is normally resolved by giving the answer set. This hides intermediate nodes visited by the query engine, which might contain sensitive information. We intend to expose this information so that it can be analysed, for example from the point of view of access control.

Example 1 Consider the family tree of Figure 1 with query_1 asking for family members with following siblings:

$$\mathsf{query}_1 = \mathsf{descendant} :: \star\,[\mathsf{following\_sibling} :: \star]$$

The answer set (i.e. Cain and Abel) does not reveal (1) the name of some of the following siblings (i.e. Seth), (2) that one of the members of the answer set is in fact a following sibling himself (i.e. Abel), and (3) the multiplicity of the answers (Cain is included for two reasons). So the answer set hides information that is available to the query engine. This information may be sensitive, and we are interested in making this information available for analysis. We achieve this by resolving a query not to the answer set but to the entire *answer trace* from the root produced by the query engine. For the example above there are three traces:

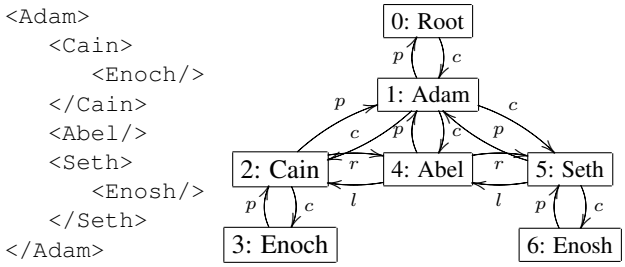

Figure 1. Sample family tree in XML format (left) and in navigational format (right).

$$\begin{aligned}\mathsf{results}_1 = \{&[\mathsf{Root},\ \mathsf{Adam},\ \mathsf{Cain},\ (\mathsf{Abel},\ \mathsf{Seth}),\ \mathsf{Cain}],\\ &[\mathsf{Root},\ \mathsf{Adam},\ \mathsf{Cain},\ (\mathsf{Abel}),\ \mathsf{Cain}],\\ &[\mathsf{Root},\ \mathsf{Adam},\ \mathsf{Abel},\ (\mathsf{Seth}),\ \mathsf{Abel}]\}\end{aligned}$$

Some tags, like Abel and Seth in the first trace, are shown in parentheses to indicate that they are the result of exploring the predicate $[\mathsf{following\_sibling} :: \star]$ of query_1. Other tags, such as Cain, are shown twice in the first trace because they have been visited twice: firstly while moving right from Adam to Cain and secondly when returning from Seth to Cain.

We can now use the information contained in a trace for analysis purposes, such as access control. We give two examples. Firstly, suppose that (if only for historical reasons) the node tagged Cain should not be included in a trace that contains Abel also. Furthermore, we should like to be free to choose whether to access Cain first, or whether to access Abel first. This corresponds to (the object specification of) a Chinese wall policy, where Cain and Abel are in the same conflict of interest class. XPath is not powerful enough to formulate such a *general* policy because we do not know a-priori which axes to navigate to travel between members of a conflict of interest class. All we could hope to do is to formulate a *specific* policy for each query. To solve this problem we use Linear Temporal Logic (LTL) to express the policy as follows (for generality extending the conflict

Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)
1530-1311/05 $20.00 © 2005 **IEEE**

of interest class to all children of Adam):

$$\mathsf{Chinese\_wall} = \Box(\ \mathsf{Cain}\ \rightarrow\ \neg\ \Diamond(\ \mathsf{Abel}\ \vee\ \ \mathsf{Seth}))$$

The formula Chinese_wall states that we should always (operator $\Box$) have that as soon as we encounter Cain, then we must not eventually (operator $\Diamond$) encounter either Abel or Seth. This corresponds to the mandatory aspect of the Chinese wall policy. The formula Chinese_wall does not insist that Cain is ever encountered, which corresponds to the discretionary aspect of the Chinese wall policy.

Secondly, using an idea of de Alfaro [9], suppose that every trace to a confidential node Cain must pass through an access control node Adam, thus blocking access via Abel. This can be formalised intuitively in LTL with past operators (an equivalent LTL expression with only future operators exists but it is less intuitive):

$$\mathsf{Access\_control} = \Box(\ \mathsf{Cain}\ \rightarrow\ \Diamond^{-1}\ \mathsf{Adam})$$

The formula Access_control states that any access of Cain is due to some earlier access of Adam.

Having motivated using LTL to express access control policies, a natural target for expressing a query is also LTL, so that we can combine them simply with a logical $\wedge$ operator, using the same formalism and implementation for both querying and access control. Therefore, the focus of the paper is on the semantics of positive core XPath because this can be translated efficiently into LTL. The main contributions are (1) a novel trace semantics for positive core XPath, (2) the translation of positive core XPath into LTL, (3) the correctness proof of the translation with respect to the trace semantics, and (4) a proof of concept implementation of the system.

The next section discusses related work. Section 3 motivates the positive core XPath subset. Section 4 formalises undecorated XML trees. Section 5 defines the Kripke structure that forms the link between the formalised XML tree representation and the semantics of LTL. Section 6 defines the embedding of positive core XPath into LTL via a translation algorithm. Section 7 provides a natural semantics for positive core XPath. In the Technical Report version [17] of this paper we provide a correctness proof of the translation with respect to the natural semantics. Section 8 presents the implementation of the positive core XPath engine using the SPIN model checker [18], and compares the performance of the implementation to that of two existing XPath query engines. The last Section concludes and suggests future work.

2. Related work

Our work has similarities with the work of Afanasiev et al [3, 2], who translate XPath into the existential fragment of Computation Tree Logic (CTL), using the nuSMV model checker as the query engine. The differences include: (1) we are interested in trace semantics, whereas Afanasiev et al work with the standard semantics for answer sets; (2) our method of model building is orders of magnitude more efficient. The efficiency of our method is mainly due to the judicious use of Embedded C code support provided by the SPIN model checker. Since SPIN only supports LTL, we represent XPath queries using LTL, rather than CTL.

XML based access control offers three fundamental choices [21]. Should the XML data be filtered according to the access control policy: (1) before a query is applied, (2) after the query is applied, or (3) should the query be rewritten? Security views are an example of case (1). However, security views are expensive to compute and to maintain, which is why Fan et al [10] propose a method of avoiding to build security views using efficient query optimization techniques. Our approach to combining a query with an access control object specification is an example of case (3): the model checker ensures that only relevant parts of the state space are explored. Luo et al [21] also perform query rewriting, but consider forward axes only.

Fundulaki and Marx [12] use XPath to represent the object specification of an access control policy, which is less powerful than using LTL for the same purpose.

Fu et al [11] use SPIN to model check XPath queries but their approach is radically different from ours in the sense that both the XML data and the query are part of the model. Fu et al use LTL formulae to specify liveness properties of the model, where we use LTL for the queries. Fu et al do not present performance data.

3. Positive core XPath

Full XPath is impractical to use as a tool for investigating the fundamental relation between query and access control. Several subsets have been defined, such as Core XPath [14], Simple XPath [3], and Navigational XPath [22]. We adopt a similar approach in that we (1) omit expressions and focus on location paths and predicates, and (2) support most (11 of the 13) axes, omitting attribute and namespace only. We omit negations for reasons to be explained later. Our subset is essentially positive core XPath, which is core XPath without negations. The abstract syntax of positive core XPath is:

$$\begin{aligned}
&\mathbb{X} \equiv \mathbb{X} \parallel \mathbb{X} \mid / \mathbb{X} \mid \mathbb{X} / \mathbb{X} \mid \mathbb{X}[\mathbb{Q}] \mid \mathbb{A} :: \mathbb{L} \\
&\mathbb{Q} \equiv \mathbb{X} \\
&\mathbb{A} \equiv \mathsf{self} \mid \\
&\qquad \mathsf{child} \mid \mathsf{descendant} \mid \mathsf{descendant_or_self} \mid \\
&\qquad \mathsf{parent} \mid \mathsf{ancestor} \mid \mathsf{ancestor_or_self} \mid \\
&\qquad \mathsf{preceding_sibling} \mid \mathsf{following_sibling} \mid \\
&\qquad \mathsf{preceding} \mid \mathsf{following}
\end{aligned}$$

The node test $\mathbb{L}$ in a step $\mathbb{A} :: \mathbb{L}$ is restricted to a name test (i.e. kind tests are not supported, which is consistent with the use of undecorated XML data). We use location paths $\mathbb{X}$ by way of predicates $\mathbb{Q}$.

3.1. Disjunction

A typical answer contains several results. Hence we should expect the trace semantics of a query to be a set of traces. The semantics of the $\|$ operator applied to two queries is therefore the union of the traces returned for each query separately.

Example 2 Consider query_2 below, which in the standard semantics yields an answer set consisting of Cain and Seth:

$$\begin{aligned}\mathsf{query}_2 = \mathsf{descendant} &:: \star \\ &[\mathsf{child} :: \mathsf{Enoch} \vee \mathsf{child} :: \mathsf{Enosh}]\end{aligned}$$

The standard semantics for XPath prescribes that the result of the predicate should be a Boolean. In our interpretation we take an empty trace to mean false and a non-empty trace to represent true [26]. However, we should also like to preserve the traces resulting from the predicate, because all visited nodes must be kept for further analysis. This leads to the idea that the result should consist of two traces, both with an initial segment corresponding to $\mathsf{descendant} :: \star$. Then the traces differ: one contains the trace corresponding to the left hand side of the $\vee$ operator, and the other takes care of the right hand side. In both cases a common trailing segment follows. The initial and trailing segment are effectively copied and concatenated to each intermediate segment, yielding the following result:

$$\begin{aligned}\mathsf{results}_2 = \{&[\mathsf{Root}, \mathsf{Adam}, \mathsf{Cain}, (\mathsf{Enoch}), \mathsf{Cain}], \\ &[\mathsf{Root}, \mathsf{Adam}, \mathsf{Seth}, (\mathsf{Enosh}), \mathsf{Seth}]\}\end{aligned}$$

With this "copying" semantics in mind the $\vee$ and $\|$ operators are identified, thus obviating the need for a separate $\vee$ operator [4, Proposition 2]. So query_2 is interpreted as:

$$\begin{aligned}\mathsf{query}_{2'} = \mathsf{descendant} &:: \star \\ &[\mathsf{child} :: \mathsf{Enoch} \,\|\, \mathsf{child} :: \mathsf{Enosh}]\end{aligned}$$

3.2. Conjunction

The trace semantics of a location path with a predicate is the concatenation of the trace of the location path and the trace of the predicate.

Example 3 Consider query_3, which in the standard semantics returns the singleton answer set $\{\mathsf{Adam}\}$:

$$\begin{aligned}\mathsf{query}_3 = \mathsf{descendant} &:: \star \\ &[\mathsf{child} :: \mathsf{Cain} \wedge \mathsf{child} :: \mathsf{Abel}]\end{aligned}$$

The resulting trace should contain an initial segment corresponding to the location path $\mathsf{descendant} :: \star$. However for the predicate to succeed we must be sure that there is at least one non-empty trace corresponding to the left hand side of the $\wedge$ operator as well as a non-empty trace corresponding to the right hand side. Both non-empty traces must be returned as part of the full trace, which we achieve by concatenating the results. We have arbitrarily chosen to concatenate the right hand side trace onto the left hand side trace; interleaving or reordering would also be possible but this is subject to further work. In all cases a trailing segment will follow. The result then becomes:

$$\mathsf{result}_3 = [\mathsf{Root}, \mathsf{Adam}, (\mathsf{Cain}), (\mathsf{Abel}), \mathsf{Adam}]$$

This, however, is exactly the trace that would be returned by the following query:

$$\begin{aligned}\mathsf{query}_{3'} = \mathsf{descendant} &:: \star \\ &[\mathsf{child} :: \mathsf{Cain}][\mathsf{child} :: \mathsf{Abel}]\end{aligned}$$

Therefore we can dispense with the $\wedge$ operator also, as repeated use of predicates can achieve the desired effect [4, Proposition 2]. Propositions with $\vee$, $\wedge$ and $\neg$ can always be written in conjunctive normal form [20]; we can thus remove all nested conjunctions and disjunctions.

3.3. Negation

Negation is a problem because it is unclear what trace to return for a negated predicate. Assume that predicates are in conjunctive normal form, and that all occurrences of $\wedge$ and $\vee$ have been removed as described above. Then there are only, possibly negated, atomic propositions left.

Example 4 Consider query_4, which in the standard semantics returns the singleton answer set $\{\mathsf{Root}\}$:

$$\mathsf{query}_4 = \mathsf{descendant\_or\_self} :: \star[\neg\, \mathsf{parent} :: \star]$$

The question now is: which traces(s) to return for the predicate? (a) Should it be all possible traces that do not satisfy the predicate? This would be infinitely many with the 11 axes of XPath! (b) Or should the trace be empty? This would jeopardise our ability to analyse the trace properly: Consider our family tree again with a query asking for brothers not involved in fratricide. Should the query return $\{\mathsf{Seth}\}$? Or should it return an empty answer set because it violates the Chinese wall policy? (c) The most likely possibility is to label segments of the trace that correspond to negated steps, so that these can be distinguished from positive steps in the analysis. This, however, we leave as future work and for now omit negation. Note that often in policy specifications the same approach is taken: what is not explicitly allowed is forbidden, hence negative steps are not

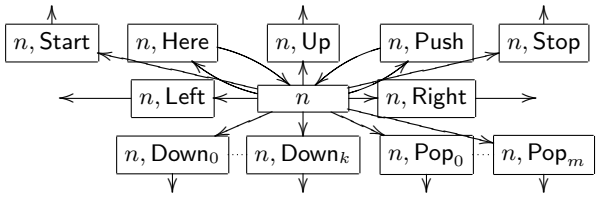

Figure 2. State n showing the nine possible directions for reaching a successor state.

always necessary [16]. However, see Section 8 for an experiment with alternative (a) in SPIN. This concludes the motivation of the positive core XPath subset.

4. XML data representation

XPath queries operate on an appropriate representation of the data that we assume to be bulk loaded; dealing with updates and inserts is beyond the scope of the paper. To provide efficient support for the 11 axes in queries, a representation of the XML data is needed that is slightly more sophisticated than a tree. Figure 1 (right) shows the navigational representation that we adopt. The four types of edges shown are p for parent, c for child, r for *immediate* following_sibling and l for *immediate* preceding_sibling. All other axes (except self) are supported by traversing more than one edge.

The nodes of the graph are represented by a given set $\mathbb{N}$, which in the case of our running example is:

$$\mathbb{N} \equiv \mathsf{Root} \mid \mathsf{Adam} \mid \mathsf{Cain} \mid \mathsf{Enoch} \mid \mathsf{Abel} \mid \mathsf{Seth} \mid \mathsf{Enosh}$$

The edges are represented by four functions, one for each type of edge (i.e. $\mathsf{up_d}$, $\mathsf{down_d}$, $\mathsf{left_d}$, and $\mathsf{right_d}$). In addition we need a function to return to the root ($\mathsf{root_d}$), as well as a function to stay put ($\mathsf{here_d}$). We show only the definition of $\mathsf{down_d}$, the remaining functions are similar.

$$\begin{array}{ll}
\mathsf{down_d} & :: \mathbb{N} \rightarrow \{\mathbb{N}\} \\
\mathsf{down_d}(\mathsf{Root}) & = \{\mathsf{Adam}\} \\
\mathsf{down_d}(\mathsf{Adam}) & = \{\mathsf{Cain},\ \mathsf{Abel},\ \mathsf{Seth}\} \\
\mathsf{down_d}(\mathsf{Cain}) & = \{\mathsf{Enoch}\} \\
\mathsf{down_d}(\mathsf{Seth}) & = \{\mathsf{Enosh}\} \\
\mathsf{down_d}\ _ & = \{\}
\end{array}$$

We follow the approach of the work cited at the beginning of Section 3 to focus on the tags of XML data, omitting all other information, so that this concludes the presentation of our representation of an undecorated XML tree.

5. Kripke structure

Before we give the translation of XPath into LTL we develop a Kripke structure for the resulting logic. The structure is based on the definition of two sets: $\mathbb{N}$, given earlier for the nodes, and $\mathbb{D}$ for the directions corresponding to the axes (Here for self, Up for parent, Down for child, Left for immediate preceding_sibling, and Right for immediate following_sibling) as well as a further four directions (Start, Stop, Push, and Pop) discussed below.

$$\begin{array}{l}
\mathbb{D} \equiv \mathsf{Start} \mid \mathsf{Here} \mid \mathsf{Up} \mid \mathsf{Down} \mid \mathsf{Left} \mid \mathsf{Right} \mid \\
\qquad \mathsf{Push} \mid \mathsf{Pop} \mid \mathsf{Stop}
\end{array}$$

Figure 2 shows all nodes in the Kripke structure that correspond to a single node of an XML tree. This representation is quadratic in the number of nodes of the original XML tree, which is clearly inefficient. We will come back to this issue in Section 8, but we need to make the situation worse first by considering how to deal with predicates. Referring back to the introduction, we saw that a predicate yields a trace segment that returns to the starting node of the segment, to linearise a finite branching structure. To support this we need a stack of nodes in the Kripke structure. The states of the Kripke structure are then defined by the triple $\mathbb{S}$ below, where $\overline{\mathbb{N}}$ represents the stack (i.e. a list of nodes):

$$\begin{array}{l}
\mathbb{S} \equiv (\mathbb{N},\ \mathbb{D},\ \overline{\mathbb{N}}) \\
\overline{\mathbb{N}} \equiv [\mathbb{N}]
\end{array}$$

Given an XML tree with n nodes, and a query with predicates nested to a depth of d, the state space in the worst case grows as $n^{(d+1)}$. In practice the state space remains small as we shall see later (Section 8).

We now have all ingredients to show the Kripke structure μ of our running example below.

$$\begin{array}{lll}
\mathbb{M}\alpha \equiv (\{\alpha\},\ \alpha \rightarrow \{\alpha\},\ \alpha \rightarrow \{\mathbb{L}\}) & & \\
\mu & :: \mathbb{M}\,\mathbb{S} & \\
\mu & = (\{(\mathsf{n},\ \mathsf{d},\ \mathsf{s}) \mid \mathsf{n} \in \mathbb{N} \wedge\ \mathsf{d} \in \mathbb{D} \wedge\ \mathsf{s} \in \overline{\mathbb{N}}\},\ \sigma,\ \lambda) & \\
 & \textbf{where} & \\
 & \sigma(\mathsf{n},\ \mathsf{Start},\ \mathsf{s}) & = \{(\mathsf{root_d}\ \mathsf{n},\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Here},\ \mathsf{s}) & = \{(\mathsf{n},\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Up},\ \mathsf{s}) & = \{(\mathsf{n}',\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{n}' \in \mathsf{up_d}\ \mathsf{n} \wedge\ \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Down},\ \mathsf{s}) & = \{(\mathsf{n}',\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{n}' \in \mathsf{down_d}\ \mathsf{n} \wedge\ \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Left},\ \mathsf{s}) & = \{(\mathsf{n}',\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{n}' \in \mathsf{left_d}\ \mathsf{n} \wedge\ \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Right},\ \mathsf{s}) & = \{(\mathsf{n}',\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{n}' \in \mathsf{right_d}\ \mathsf{n} \wedge\ \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Push},\ \mathsf{s}) & = \{(\mathsf{n},\ \mathsf{d}',\ \mathsf{n} : \mathsf{s}) \mid \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Pop},\ \mathsf{n}' : \mathsf{s}) & = \{(\mathsf{n}',\ \mathsf{d}',\ \mathsf{s}) \mid \mathsf{d}' \in \mathbb{D}\} \\
 & \sigma(\mathsf{n},\ \mathsf{Pop},\ []) & = \{\} \\
 & \sigma(\mathsf{n},\ \mathsf{Stop},\ \mathsf{s}) & = \{\} \\
 & \lambda(\mathsf{n},\ \mathsf{d},\ \mathsf{s}) & = \{\mathsf{n},\ \mathsf{d}\}
\end{array}$$

Here the function $\sigma(\mathsf{n}, \mathsf{d}, \mathsf{s})$ defines the set of all possible successor states. For example the successor state of

Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)
1530-1311/05 $20.00 © 2005 **IEEE**

$(\mathsf{n}, \mathsf{Push}, \mathsf{s})$ consists of the set of states $(\mathsf{n}, \mathsf{d}', \mathsf{n} : \mathsf{s})$, where d' ranges over all (9) directions in $\mathbb{D}$, and where $\mathsf{n} : \mathsf{s}$ represents the current stack s extended with the current node n. Summarising, the interpretation of state $(\mathsf{n}, \mathsf{d}, \mathsf{s})$ is: we are now at node n going in the direction d, with current stack s. The function $\lambda(n, d, s)$ defines the atomic propositions for the state (n, d, s). We have tacitly assumed here that all nodes in the tree have a unique tag. If this is not the case, the Kripke structure must be extended with a unique identifier for each node. We will ensure that this is the case in the high performance SPIN models.

This concludes the presentation of the Kripke structure so that we can turn our attention to the translation of positive core XPath into LTL.

6. Translation of positive core XPath into LTL

The function $\mathcal{T}_\mathsf{x}$ below translates an XPath query into an LTL formula. The function takes a query as its first argument, and an LTL formula ϕ which represents what should happen after we have dealt with the query. Consider for example the first clause of $\mathcal{T}_\mathsf{x}$. Since ϕ represents what happens after $\mathsf{xp}_1 \| \mathsf{xp}_2$, ϕ must happen after xp_1 as well as xp_2. This corresponds to the "copying" semantics alluded to in the introduction.

Consider also the second clause, which states that for an absolute query /xp we go from the current node in the Start direction, leading to the node Root in the next (X) step. Then we continue with xp, ultimately followed by ϕ. The remaining clauses are intended to be self explanatory.

$$
\begin{array}{ll}
\mathcal{T}_\mathsf{x} & :: \mathbb{X} \rightarrow \mathbb{T} \rightarrow \mathbb{T} \\
\mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1 \parallel \mathsf{xp}_2]\!]\phi & = \mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1]\!]\phi \vee \mathcal{T}_\mathsf{x}[\![\mathsf{xp}_2]\!]\phi \\
\mathcal{T}_\mathsf{x}[\![/\ \mathsf{xp}]\!]\phi & = \mathsf{Start} \wedge \mathsf{X}(\mathsf{Root} \wedge \mathcal{T}_\mathsf{x}[\![\mathsf{xp}]\!]\phi) \\
\mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1\ /\ \mathsf{xp}_2]\!]\phi & = \mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1]\!](\mathcal{T}_\mathsf{x}[\![\mathsf{xp}_2]\!]\phi) \\
\mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1[\mathsf{xp}_2]]\!]\phi & = \mathcal{T}_\mathsf{x}[\![\mathsf{xp}_1]\!](\mathsf{Push} \wedge \\
 & \quad \mathsf{X}(\mathcal{T}_\mathsf{x}[\![\mathsf{xp}_2]\!](\mathsf{Pop} \wedge \mathsf{X}\,\phi))) \\
\mathcal{T}_\mathsf{x}[\![\mathsf{a} :: \mathsf{l}]\!]\phi & = \mathcal{T}_\mathsf{a}[\![\mathsf{a}]\!](\mathsf{l} \wedge \phi)
\end{array}
$$

The function $\mathcal{T}_\mathsf{a}$ below follows the same pattern as $\mathcal{T}_\mathsf{x}$. The first argument is an axis and the second argument ϕ corresponds to the query that must be matched after the current axis has been matched. For example the first clause states that the proposition Here must be true in the current state, and that ϕ must hold in the next state.

Also note the difference between descendant and descendant_or_self. In the former we check first that a move in the direction Down can be made, optionally followed by a further sequence of moves in the Down direction until finally a state is found in which ϕ is true. In the latter case we accept either a move to the current node (direction Here) or the moves implied by the axis descendant. The cases for the remaining axes are expected to be self explanatory.

$$
\begin{array}{ll}
\mathcal{T}_\mathsf{a} & :: \mathbb{A} \rightarrow \mathbb{T} \rightarrow \mathbb{T} \\
\mathcal{T}_\mathsf{a}[\![\mathsf{self}]\!]\phi & = \mathsf{Here} \wedge \mathsf{X}\,\phi \\
\mathcal{T}_\mathsf{a}[\![\mathsf{child}]\!]\phi & = \mathsf{Down} \wedge \mathsf{X}\,\phi \\
\mathcal{T}_\mathsf{a}[\![\mathsf{parent}]\!]\phi & = \mathsf{Up} \wedge \mathsf{X}\,\phi \\
\mathcal{T}_\mathsf{a}[\![\mathsf{descendant}]\!]\phi & = \mathsf{Down} \wedge \mathsf{X}(\mathsf{Down}\ \mathsf{U}\ \phi) \\
\mathcal{T}_\mathsf{a}[\![\mathsf{ancestor}]\!]\phi & = \mathsf{Up} \wedge \mathsf{X}(\mathsf{Up}\ \mathsf{U}\ \phi) \\
\mathcal{T}_\mathsf{a}[\![\mathsf{descendant\_or\_self}]\!]\phi & = \mathcal{T}_\mathsf{a}[\![\mathsf{self}]\!]\phi \vee \mathcal{T}_\mathsf{a}[\![\mathsf{descendant}]\!]\phi \\
\mathcal{T}_\mathsf{a}[\![\mathsf{ancestor\_or\_self}]\!]\phi & = \mathcal{T}_\mathsf{a}[\![\mathsf{self}]\!]\phi \vee \mathcal{T}_\mathsf{a}[\![\mathsf{ancestor}]\!]\phi \\
\mathcal{T}_\mathsf{a}[\![\mathsf{following\_sibling}]\!]\phi & = \mathsf{Right} \wedge \mathsf{X}(\mathsf{Right}\ \mathsf{U}\ \phi) \\
\mathcal{T}_\mathsf{a}[\![\mathsf{preceding\_sibling}]\!]\phi & = \mathsf{Left} \wedge \mathsf{X}(\mathsf{Left}\ \mathsf{U}\ \phi) \\
\mathcal{T}_\mathsf{a}[\![\mathsf{following}]\!]\phi & = \mathsf{Up}\ \mathsf{U}(\mathsf{Right} \wedge \\
 & \quad \mathsf{X}(\mathsf{Right}\ \mathsf{U}(\mathsf{Down}\ \mathsf{U}\ \phi))) \\
\mathcal{T}_\mathsf{a}[\![\mathsf{preceding}]\!]\phi & = \mathsf{Up}\ \mathsf{U}(\mathsf{Left} \wedge \\
 & \quad \mathsf{X}(\mathsf{Left}\ \mathsf{U}(\mathsf{Down}\ \mathsf{U}\ \phi)))
\end{array}
$$

We present some examples of the translation.

Example 5 Query_5 delivers the traces from the current context node to a child with tag Adam. There is one such trace from the Root.

$$
\begin{array}{ll}
\mathsf{query}_5 & = \mathsf{child} :: \mathsf{Adam} \\
\mathsf{ltl}_5 & = \mathcal{T}_\mathsf{x}[\![\mathsf{query}_5]\!]\ \mathsf{Stop} \\
 & = \mathsf{Down} \wedge \mathsf{X}(\ \mathsf{Adam} \wedge \ \mathsf{Stop}) \\
\mathsf{result}_5 & = [(\mathsf{Root},\ \mathsf{Down},\ []),\ (\mathsf{Adam},\ \mathsf{Stop},\ [])]
\end{array}
$$

The LTL translation ltl_5 is valid on the trace result_5: $\mathsf{result}_5 \models \mathsf{ltl}_5$ □

Example 1 revisited We now revisit query_1 to demonstrate how predicates are translated.

$$
\begin{array}{l}
\mathsf{ltl}_1 = \mathcal{T}_\mathsf{x}[\![\mathsf{query}_1]\!]\ \mathsf{Stop} \\
\quad = \mathsf{Down} \wedge \mathsf{X}(\ \mathsf{Down}\ \mathsf{U}(\ \mathsf{Push} \wedge \\
\qquad \mathsf{X}(\ \mathsf{Right} \wedge \mathsf{X}(\ \mathsf{Right}\ \mathsf{U}(\ \mathsf{Pop} \wedge \mathsf{X}\ \ \mathsf{Stop})))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{results}_1 = \{[(\mathsf{Root},\ \mathsf{Down},\ []),\ (\mathsf{Adam},\ \mathsf{Down},\ []), \\
\qquad (\mathsf{Cain},\ \mathsf{Push},\ []),\ (\mathsf{Cain},\ \mathsf{Right},\ [\mathsf{Cain}]), \\
\qquad (\mathsf{Abel},\ \mathsf{Right},\ [\mathsf{Cain}]),\ (\mathsf{Seth},\ \mathsf{Pop},\ [\mathsf{Cain}]), \\
\qquad (\mathsf{Cain},\ \mathsf{Stop},\ [])], \\
\qquad [(\mathsf{Root},\ \mathsf{Down},\ []),\ (\mathsf{Adam},\ \mathsf{Down},\ []), \\
\qquad (\mathsf{Cain},\ \mathsf{Push},\ []),\ (\mathsf{Cain},\ \mathsf{Right},\ [\mathsf{Cain}]), \\
\qquad (\mathsf{Abel},\ \mathsf{Pop},\ [\mathsf{Cain}]),\ (\mathsf{Cain},\ \mathsf{Stop},\ [])], \\
\qquad [(\mathsf{Root},\ \mathsf{Down},\ []),\ (\mathsf{Adam},\ \mathsf{Down},\ []), \\
\qquad (\mathsf{Abel},\ \mathsf{Push},\ []),\ (\mathsf{Abel},\ \mathsf{Right},\ [\mathsf{Abel}]), \\
\qquad (\mathsf{Seth},\ \mathsf{Pop},\ [\mathsf{Abel}]),\ (\mathsf{Abel},\ \mathsf{Stop},\ [])]\}
\end{array}
$$

The LTL formula ltl_1 is valid on all traces of the set $\mathsf{results}_1$: $\bigwedge\{\mathsf{r} \models \mathsf{ltl}_1 \mid \mathsf{r} \in \mathsf{results}_1\}$ □

This concludes the LTL translation of positive core XPath.

Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)
1530-1311/05 $20.00 © 2005 **IEEE**

7. Natural semantics for XPath

Borrowing ideas from Wadler [27], we define the semantics of positive core XPath as a relation between a trace and a query on the left hand side and a trace on the right hand side. The trace on the left hand side is the end point of the current trace, from which the current (context) node can be found. Consider for example the rule [abs] for absolute queries. The endpoint of the current trace is $(x, \bot)$, where x is the current context node, and the direction in which to go is yet unknown ($\bot$). The premise of the rule asserts that the relative query xp started at the Root yields a trace xs', where the direction taken from the Root will be known (i.e. $\neq \bot$). The right hand side of the conclusion prepends the state (x, Start) to xs', to account for the fact that now we know in which direction to proceed from the original, initial node x. We hope that the remaining clauses are self explanatory. (As usual we omit explicit coercions, for example using the : operator for the concatenation of traces and traces, traces and elements etc.).

$$
\begin{array}{lll}
\mathbb{P} & \equiv & [(\mathbb{N},\ \mathbb{D})] \\
\rightarrow & :: & \langle \mathbb{P},\ \mathbb{X} \rangle \leftrightarrow \mathbb{P} \\
[\mathsf{bar}^1] & & \dfrac{\langle (x,\ \bot),\ xp_1 \rangle \rightarrow xs'}{\langle (x,\ \bot),\ xp_1 \parallel xp_2 \rangle \rightarrow xs'} \\
[\mathsf{bar}^2] & & \dfrac{\langle (x,\ \bot),\ xp_2 \rangle \rightarrow xs'}{\langle (x,\ \bot),\ xp_1 \parallel xp_2 \rangle \rightarrow xs'} \\
[\mathsf{abs}] & & \dfrac{\langle (\mathsf{Root},\ \bot),\ xp \rangle \rightarrow xs'}{\langle (x,\ \bot),\ /\ xp \rangle \rightarrow (x,\ \mathsf{Start}) : xs'} \\
[\mathsf{slash}] & & \dfrac{\langle (x,\ \bot),\ xp_1 \rangle \rightarrow xs' : (x',\ \bot),\ \langle (x',\ \bot),\ xp_2 \rangle \rightarrow xs''}{\langle (x,\ \bot),\ xp_1 / xp_2 \rangle \rightarrow xs' : xs''} \\
[\mathsf{pred}] & & \dfrac{\langle (x,\ \bot),\ xp_1 \rangle \rightarrow xs' : (x',\ \bot),\ \langle (x',\ \bot),\ xp_2 \rangle \rightarrow xs'' : (x'',\ \bot)}{\langle (x,\ \bot),\ xp_1[xp_2] \rangle \rightarrow (xs' : (x',\ \mathsf{Push}) : xs'' : (x'',\ \mathsf{Pop}) : (x',\ \bot)} \\
[\mathsf{step}] & & \dfrac{xs' : (x',\ \bot) \in \mathcal{P}_a[\![a]\!](x,\ \bot)}{\langle (x,\ \bot),\ a :: l \rangle \rightarrow xs' : (x',\ \bot),\ \textbf{if}\ l = \star \vee l = x'}
\end{array}
$$

The semantic function $\mathcal{P}_x$ below provides a convenient interface to the natural semantics.

$$
\begin{array}{lll}
\mathcal{P}_x & :: & \mathbb{X} \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
\mathcal{P}_x[\![xp]\!][(x,\ \bot)] & = & \{xs' : (x',\ \mathsf{Stop}) \mid xs' : (x',\ \bot) \in \langle (x,\ \bot),\ xp \rangle \rightarrow\}
\end{array}
$$

The rule [step] relies on the function $\mathcal{P}_a$ below to deal with the 11 axes of XPath.

$$
\begin{array}{lll}
\mathcal{P}_a & :: & \mathbb{A} \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
\mathcal{P}_a[\![\mathsf{self}]\!] & = & \mathsf{here}_p \\
\mathcal{P}_a[\![\mathsf{child}]\!] & = & \mathsf{down}_p \\
\mathcal{P}_a[\![\mathsf{parent}]\!] & = & \mathsf{up}_p \\
\mathcal{P}_a[\![\mathsf{descendant}]\!] & = & \mathsf{down}_p +_p \\
\mathcal{P}_a[\![\mathsf{ancestor}]\!] & = & \mathsf{up}_p +_p \\
\mathcal{P}_a[\![\mathsf{descendant\_or\_self}]\!] & = & \mathcal{P}_a[\![\mathsf{self}]\!] \vee_p \mathcal{P}_a[\![\mathsf{descendant}]\!] \\
\mathcal{P}_a[\![\mathsf{ancestor\_or\_self}]\!] & = & \mathcal{P}_a[\![\mathsf{self}]\!] \vee_p \mathcal{P}_a[\![\mathsf{ancestor}]\!] \\
\mathcal{P}_a[\![\mathsf{following\_sibling}]\!] & = & \mathsf{right}_p +_p \\
\mathcal{P}_a[\![\mathsf{preceding\_sibling}]\!] & = & \mathsf{left}_p +_p \\
\mathcal{P}_a[\![\mathsf{following}]\!] & = & \mathsf{horizontal}_p\ \mathsf{right}_p \\
\mathcal{P}_a[\![\mathsf{preceding}]\!] & = & \mathsf{horizontal}_p\ \mathsf{left}_p
\end{array}
$$

The function $\mathcal{P}_a$ in turn relies on a number of functions below to calculate the possible traces from the current node (again found in the endpoint of the current trace) in the direction indicated by the axis. For example down_p with a current node x yields a set of segments $(x, \mathsf{Down}) : (y, \bot)$ where y ranges over all children of node x, as defined by the function down_d of Section 4. The result of down_d is empty if node x has no children.

$$
\begin{array}{lll}
\mathsf{go} & :: & \mathbb{D} \rightarrow (\mathbb{N} \rightarrow \{\mathbb{N}\}) \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
\mathsf{go\ d\ f}(x,\ \bot) & = & \{(x,\ \mathsf{d}) : (y,\ \bot) \mid y \in \mathsf{f}\ x\} \\
\mathsf{here}_p & :: & \mathbb{P} \rightarrow \{\mathbb{P}\} \\
\mathsf{here}_p & = & \mathsf{go\ Here\ here}_d \\
\mathsf{down}_p,\ \mathsf{up}_p & :: & \mathbb{P} \rightarrow \{\mathbb{P}\} \\
\mathsf{down}_p & = & \mathsf{go\ Down\ down}_d \\
\mathsf{up}_p & = & \mathsf{go\ Up\ up}_d \\
\mathsf{left}_p,\ \mathsf{right}_p & :: & \mathbb{P} \rightarrow \{\mathbb{P}\} \\
\mathsf{left}_p & = & \mathsf{go\ Left\ left}_d \\
\mathsf{right}_p & = & \mathsf{go\ Right\ right}_d
\end{array}
$$

The function $\mathsf{horizontal}_p$ is used by the axes preceding and following to discover trace segments corresponding to the nodes that precede respectively follow the current node in XML document order.

$$
\begin{array}{lll}
\mathsf{horizontal}_p & :: & (\mathbb{P} \rightarrow \{\mathbb{P}\}) \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
\mathsf{horizontal}_p\ \mathsf{fp} & = & \mathsf{hhc} \vee_p ((\mathsf{up}_p +_p) \wedge_p \mathsf{hhc}) \\
 & & \textbf{where} \\
 & & \mathsf{h} = \mathsf{fp} +_p \\
 & & \mathsf{hhc} = \mathsf{h} \vee_p (\mathsf{h} \wedge_p (\mathsf{down}_p +_p))
\end{array}
$$

Finally we need three operators ($+_p$, $\wedge_p$, and $\vee_p$) to glue trace segments together.

$$
\begin{array}{lll}
+_p & :: & (\mathbb{P} \rightarrow \{\mathbb{P}\}) \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
\mathsf{r} +_p & = & \mathsf{r} \vee_p (\mathsf{r} \wedge_p \mathsf{r} +_p) \\
\wedge_p,\ \vee_p & :: & (\mathbb{P} \rightarrow \{\mathbb{P}\}) \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \rightarrow (\mathbb{P} \rightarrow \{\mathbb{P}\}) \\
(\mathsf{r} \wedge_p \mathsf{q})(x,\ \bot) & = & \{\mathsf{ys} : \mathsf{zs} \mid \mathsf{ys} : (y,\ \bot) \in \mathsf{r}(x,\ \bot) \wedge \mathsf{zs} \in \mathsf{q}(y,\ \bot)\} \\
(\mathsf{r} \vee_p \mathsf{q})(x,\ \bot) & = & \mathsf{r}(x,\ \bot) \cup \mathsf{q}(x,\ \bot)
\end{array}
$$

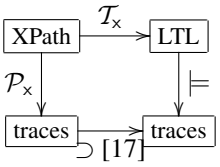

Figure 3. The relation between the translation from XPath to LTL and the semantics of both.

To conclude this section we illustrate the relation between the Natural semantics of positive core XPath and its translation into LTL using Figure 3.

8. SPIN engine

We now present two ways of representing the Kripke structure as an explicit state model for SPIN to show that in practical cases, the state space does not grow as in the worst case.

8.1. Pure Promela Model

The Promela model below is an optimised representation of the Kripke structure of Section 5. The state consists of an `mtype` declaration introducing the nodes and directions, and three variables `tag`, `dir`, and `stack` representing the current tag, direction of travel, and stack.

```
mtype={ Root, Adam, Cain, Enoch, Abel,
        Seth, Enosh, Start, Here, Up, Down,
        Left, Right, Push, Pop, Stop };
mtype tag=Root;
mtype dir=Down;
byte  stack=0;
```

The XML tree is built using a series of macros. The first parameter is the node number as shown in Figure 1, the second is the tag and the remaining parameters are the node numbers of the parent, children, and the nodes immediately to the left and the right.

```
init{
nodeR(0,Root,1);
    node3(1,Adam,0,2,4,5);
        node1r(2,Cain,1,3,4);
            node0(3,Enoch,2);
        node10r(4,Abel,1,2,5);
        node11(5,Seth,1,4,6);
            node0(6,Enosh,5);
end: skip
}
```

We do not give the definitions of the macros as these are largely repetitive. Instead we show the expansion of the node with tag `Adam`. Starting at label `s1`, where `1` is the node number of `Adam`, there is a non-deterministic choice leading to all possible successor states of `s1`. Promela does not offer a "computed goto", so this has to be simulated for popping the stack. Promela models must be finite. Therefore, we limit the stack depth to 2, supporting a nesting level of 2 for predicates. (Using qualifier flattening [23] a nesting depth of 1 would be sufficient).

```
s1: if
 :: d_step{ tag=Adam; dir=Start }; goto s0
 :: d_step{ tag=Adam; dir=Here  }; goto s1
 :: d_step{ tag=Adam; dir=Up    }; goto s0
 :: d_step{ tag=Adam; dir=Down  }; goto s2
 :: d_step{ tag=Adam; dir=Down  }; goto s4
 :: d_step{ tag=Adam; dir=Down  }; goto s5
 :: d_step{ tag=Adam; dir=Push;
    stack=(stack<<4)|1 }; goto s1
 :: d_step{ (stack&15)==0 -> tag=Adam;
    dir=Pop; stack=(stack>>4) }; goto s0
 :: ...
 :: d_step{ (stack&15)==6 -> tag=Adam;
    dir=Pop; stack=(stack>>4) }; goto s6
 :: d_step{ tag=Adam; dir=Stop  }; goto end
 fi ;
```

8.2. Promela model with Embedded C code

Promela provides facilities to embed C code in the model [19]. We use this facility to separate parsing an XML file, and building an in-memory data structure in C on the one hand from the query processing with SPIN on the other hand. We use the eXpat library to parse the XML data [7]. The in-memory data structure follows the navigational format as shown in Figure 1, and the Kripke structure. For each node in the tree we `malloc()` a node with the appropriate number of children using the C type definition:

```
typedef struct node* Nodeptr ;
typedef struct node {
    int tag ;
    int sz ; /* Number of children */
    Nodeptr parent, left, right ;
    Nodeptr child[sz] ;
} Node ;
```

The state of the Embedded C Promela model consists of five variables, where `tag`, `dir`, and `stack` are as in the pure Promela model. The added variable `ptr` points at the `Node` to which we are moving, and `nr` is used to index the appropriate `child`. The Promela model with Embedded C Code has fewer control states (106 versus 493 of the Pure Promela model) but it has more data states.

```
short tag ;
byte dir ;
int stack ;
c_state "Nodeptr ptr" "Global"
short nr ;
```

The `init` process below consists of the initialization where the C code which parses the XML tree is called. This is followed by a `do` statement with a non-deterministic choice for each of the nine directions, except for the `Down` direction, which has more cases to support nodes with many children efficiently. We show the cases for `Up` and one of the cases for `Down`, the remaining cases are similar.

```
init {
    ... Initialisation calling XML parser ...
    do
    :: d_step{
        c_expr{ now.ptr->parent != NULL } ->
        c_code{
            now.tag = now.ptr->tag ;
            now.dir = Up ;
            now.ptr = now.ptr->parent ;
        }
    }
    :: d_step{ c_expr{ now.ptr->sz > 0 } ->
        c_code {
            now.tag = now.ptr->sym ;
            now.dir = Down ;
            now.ptr = now.ptr->child[0] ;
        }
    }
    ... Other cases ...
}
```

With the Kripke structure in place all that remains is to add the never claim generated by SPIN for the LTL formula that represents the query. The never claim specifies undesirable behaviour and SPIN will try to find a counter example. Therefore every counter example represents a match of the query, showing the details of the trace as required.

8.3. Performance

We discuss the performance of the pure Promela model first, then compare the performance of the Promela model to that of proper XPath query engines. All our performance figures apply to a Sun SPARC Ultra-Enterprise Server running SunOS 5.8.

Pure Promela The number of control states defined by our running example μ from Section 5 is 493. The number of data states defined by `tag`, `dir` and `stack` is at least $7 \times 9 \times 7^2 = 3087$. Multiplied by the number of control states, this yields over 1.5 M states. However a small percentage of these states is explored, as is shown in the second row of Table 1. The columns correspond to the five example queries discussed earlier in the paper. The first row shows the query number, the second shows the number of states stored to find at least one trace. (SPIN does not guarantee to find all traces. For query_2 SPIN returns all (2) traces, but for query_1 only one of the three traces is found.)

Table 1. The number of states stored by SPIN for the family tree example.

SPIN version	$\mathsf{query}_{1\ldots5}$				
	1	2	3	4	5
pure	114	189	186	36	18
embedded C	36	91	46	7	5

The presence of data for query_4 in Table 1 is due to the fact that we translate the negated predicate into a negated LTL formula thus:

$$\mathcal{T}_{\mathsf{x}}[\![\mathsf{xp}_1[\neg\, \mathsf{xp}_2]]\!]\phi = \mathcal{T}_{\mathsf{x}}[\![\mathsf{xp}_1]\!](\neg(\mathsf{Push} \wedge \mathsf{X}(\mathcal{T}_{\mathsf{x}}[\![\mathsf{xp}_2]\!](\mathsf{Pop} \wedge \mathsf{X}\,\phi))))$$

This means that SPIN might try to discover infinitely many counter examples, which for the purpose of this experiment is capped at 100.

Promela with Embedded C Code The Promela model with embedded C code performs better than the pure Promela model, because only the work relevant for the query processing is exposed to the model checker, the rest is hidden in the C code. The number of states explored is shown in the second row of Table 1. The runtime of the query processing is not interesting since the XML tree corresponding to μ (Section 5) is tiny.

Comparison with XML Task Force and MacMill The third experiment repeats and extends the experiments of Afanasiev et al [3], using the standard XMark XML benchmark as the data base [25], with MacMill [6] and the XML taskforce query engine [13]. The six queries of Afanasiev et al are as follows:

$\mathsf{xmark}_1 =$ / child :: site / child :: regions / child :: africa / child :: item / child :: description / child :: parlist / child :: listitem / child :: text

$\mathsf{xmark}_2 =$ / descendant :: item / child :: description / child :: parlist / child :: listitem / child :: text

$\mathsf{xmark}_3 =$ / descendant :: item / descendant :: text

$\mathsf{xmark}_4 =$ descendant :: open_auction[child :: bidder]

$\mathsf{xmark}_5 =$ descendant :: item[child :: payment] [child :: location]

$\mathsf{xmark}_6 =$ descendant :: item[descendant :: payment]

The extension consists of the three queries below, which originate from Grust et al [15].

$\mathsf{xmark}_7 = \mathsf{descendant} :: \mathsf{open\_auction} /$
$\mathsf{descendant} :: \mathsf{description}$
$\mathsf{xmark}_8 = \mathsf{descendant} :: \mathsf{age} / \mathsf{ancestor} :: \mathsf{person}$
$\mathsf{xmark}_9 = \mathsf{descendant} :: \mathsf{open\_auction} / \mathsf{child} :: \mathsf{privacy} /$
$\mathsf{preceding\_sibling} :: \mathsf{bidder}$

The XML files generated by XMark range in size from 1.11MB to 111MB, using seven scaling parameter settings $\mathsf{f} = 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0$.

For each of the nine XPath queries and seven XML files, we measure the total time taken to read and parse the XML file and to execute the query, but excluding the time taken to compile the query. Each of the 9×7 measurements is an average of ten experiments, with a standard deviation of 8% or less. The query processing speed, defined as the size of the XML file divided by the time necessary to read and parse the XML input and to execute the query, is independent of the nine XMark queries and the seven data base sizes. The processing speed of MacMill is best with an average and standard deviation of $4885 \pm 2\%$ KB/s, for the TaskForce engine we found $2302 \pm 6\%$ KB/s, and for SPIN $2173 \pm 3\%$ KB/s. Overall we conclude that the SPIN implementation is competitive, which, going by the processing speed is MacMill : TaskForce : SPIN = 2.3 : 1.1 : 1.0

The second experiment shows that on pure query execution, MacMill is always slower than SPIN. (The TaskForce engine does not report pure query processing times, and it is not distributed in source form so we could not add this feature). MacMill is probably slower because it reports all answers; the SPIN implementation reports only one answer, thus providing it with an unfair advantage. Changing MacMill to provide only one answer proved difficult. Therefore we conduct a third and last experiment to put the SPIN implementation and MacMill on an equal footing.

The third experiment uses a worst case scenario based on a degenerate XML tree as follows:

```
<A1><A2>...<An> <B/> </An>...</A2></A1>
```

To answer the query $\mathsf{descendant} :: \mathsf{B}$ both implementations traverse all $n + 1$ nodes of the entire tree once, requiring both to perform the same amount of work. The time taken to answer the query is linear in n; MacMill $t = 24 \times n$ and SPIN $t = 38 \times n$. Times are in μSec, the results are based on averages of ten experiments per data point, n ranges from 10000 to 40000, the standard error in the coefficients is less than 2%, and small constant offsets are omitted. Again, we claim that the SPIN implementation is competitive because it runs at a little more than half the speed of MacMill.

The fact that SPIN does not report all answers is not a problem with the main results of the paper, i.e. the trace semantics and its embedding in LTL.

9. Conclusions and Future work

We define a novel trace semantics for positive core XPath that supports location paths, predicates, and 11 out of 13 axes. Expressions and negation are not currently supported. We show that positive core XPath can be translated into LTL. The translation is based on the idea that a branching structure as induced by location paths with predicates can be linearised effectively with the use of a stack. The translation is proved correct with respect to the trace semantics in the Technical Report version of this paper [17]. The trace semantics provides opportunities for analysis. We give two examples showing that enforcing access control policies amounts to model checking the conjunction of the policy and the (LTL translation of) the query. Finally the SPIN model checker is used as an efficient query engine, by providing it with a representation of an XML file and a never claim corresponding to the query translated into LTL. The performance of the SPIN implementation is comparable to that of the W3C XPath Taskforce Query engine.

Our SPIN implementation represents a successful experiment in creative laziness in the sense that we use existing tools (SPIN and the eXpat parser) for a new purpose (query processing) [24]. The necessary glue consists of a small Promela model and some C code (400 lines) that enable the model checker to traverse the XML tree, and a small compiler from XPath expressions into LTL (17 lines of Haskell). By comparison MacMill comprises about 2600 lines of Yacc and C++. The SPIN implementation has some undesirable features. In particular it stops after reporting one trace, and each new query must be compiled. This makes the current implementation unsuitable for practical use. A way forward would be to build, a query engine based on state of the art model checking technology. Instead of developing a tool from scratch one would use building blocks from a modular model checker and build an efficient special purpose tool with relative ease. This opens up a spectrum of possibilities ranging from a complete implementation from scratch (MacMill), via a partial implementation using existing model checker modules (future work) to our implementation with minimal glue.

10. Acknowledgements

Loredana Afanasiev and Massimo Franceschet provided their CTL benchmark and commented on the approach. Sandro Etalle suggested using multi-lateral security as a motivating example. Gerard Holzmann and Theo Ruys answered many SPIN questions. Theo Ruys, Maurice van Keulen and the anonymous TIME 2005 referees commented on a draft of the paper. Christoph Koch provided MacMill.

References

[1] M. Abadi. Logic in access control. In *18th Annual IEEE Symp. on Logic in Computer ScienceC (LICS)*, pages 228–233, Ottawa, Canada, Jun 2003. IEEE Computer Society Press, Los Alamitos, California.

[2] L. Afanasiev. XML query evaluation via CTL symbolic model checking. In *European Summer School in Logic, Language and Information (ESSLLI) Student Session*, pages 1–12, Nancy, France,, Aug 2004.

[3] L. Afanasiev, M. Franceschet, M. Marx, and M. de Rijke. CTL model checking for processing simple XPath queries. In *11th Int. Symp. on Temporal Representation and Reasoning (TIME)*, pages 117–124, Tatihou, France, Jul 2004. IEEE Computer Society Press, Los Alamitos, California.

[4] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *9th Int. Conf. on Database Theory (ICDT)*, volume LNCS 2572, pages 79–95, Siena, Italy, Jan 2003. Springer-Verlag, Berlin.

[5] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web*, 3(3):139–151, 2000.

[6] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *29th Int. Conf. on Very Large Data Bases (VLDB)*, pages 141–152, Berlin, Germany, Sep 2003. Morgan Kaufmann.

[7] J. Clark. *Expat XML Parser*. Open Software Technology Group, Fremont, California, Jul 2004.

[8] J. Clark and S. D. (eds.). *XML Path Language (XPath Version 1.0)*. W3C, Nov 1999.

[9] L. de Alfaro. Model checking the world wide web. In G. Berry, H. Comon, and A. Finkel, editors, *13th Int. Conf. on Computer Aided Verification (CAV)*, volume LNCS 2102, pages 337–349, Paris, France, Jul 2001. Springer-Verlag, Berlin.

[10] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In G. Weikum, A. C. König, and S. Deßloch, editors, *SIGMOD Int. Conf. on Management of Data*, pages 587–598, Paris, France, Jun 2004. ACM Press, New York.

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *13th conf. on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press, New York.

[12] I. Fundulaki and M. Marx. Specifying access control policies for XML documents. In *9th ACM Symp. on access control models and technologies*, pages 61–69, IBM, Yorktown Heights, USA, Jun 2004. ACM Press, New York.

[13] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *28th Int. Conf. on Very Large Data Bases (VLDB)*, pages 95–106, Hong Kong, China, Aug 2002. VLDB Endowment Inc.

[14] G. Gottlob, C. Koch, and R. Pichler. XPath processing in a nutshell. *SIGMOD Rec.*, 32(2):21–27, Jun 2003.

[15] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29(1):91–131, Mar 2004.

[16] J. Y. Halpern and V. Weissman. Using First-Order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 187–201, Pacific Grove, California, Jun 2003. IEEE Computer Society Press, Los Alamitos, California.

[17] P. H. Hartel. A trace semantics for positive core XPath (with proofs). Technical report TR-CTIT-05-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Jan 2005.

[18] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference manual*. Pearson Education Inc, Boston Massachusetts, 2004.

[19] G. J. Holzmann and R. Joshi. Model-Driven software verification. In S. Graf and L. Mounier, editors, *11th Int. SPIN Workshop: Model Checking Software*, volume LNCS 2989, pages 76–91, Barcelona, Spain, Apr 2004. Springer-Verlag Heidelberg.

[20] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, UK, 2004.

[21] B. Luo, D. Lee, W.-C. Lee, and P. Liu. QFilter: Fine-Grained Run-Time XML access control via NFA-based query rewriting. In *13th Conf. on Information and Knowledge Management (CIKM)*, pages 543–552, Washington D. C., Nov 2004. ACM Press, New York.

[22] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. Technical report, Univ. of Amsterdam, 2004.

[23] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In A. B. Chaudhri, R. Unland, C. Djeraba, and W. Lindner, editors, *XML-Based Data Management and Multimedia Engineering (EDBT)*, volume LNCS 2490, pages 109–127, Prague, Czech Republic, Mar 2002. Springer-Verlag, Heidelberg.

[24] T. C. Ruys. Optimal scheduling using branch and bound with SPIN 4.0. In T. Ball and S. K. Rajamani, editors, *10th Int. SPIN Workshop on Model Checking Software*, volume LNCS 2648, pages 1–17, Portland, Oregon, May 2003. Springer-Verlag, Berlin.

[25] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *28th Int. Conf. on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, Aug 2002. VLDB Endowment Inc.

[26] P. L. Wadler. How to replace failure by a list of successes, a method for exception handling, backtracking and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture (FPCA)*, volume LNCS 201, pages 113–128, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

[27] P. L. Wadler. Two semantics for XPath. Technical note, Dept. of Comp. Sci, Univ. of Edinburgh, Jan 2000.