A Clausal View for Access Control and XPath Query Evaluation

Barbara Fila, Siva Anantharaman

LIFO - Université d'Orléans (France),
e-mail: {fila, siva}@univ-orleans.fr

Abstract. Any positive XPath query Q (in a suitable format) can be evaluated on any given XML-like document t, under unambiguous runs of a transition system S_Q that we associate with Q. The transitions of S_Q on t are expressed as clauses. Query evaluation can be subject to access control policies enforced on the documents; the policies themselves are formulated as clauses, possibly also subject to constraints.
Keywords: XML, XPath, query, access control, constraints, clauses.

1 Introduction

Our objective in this paper is to show that XPath query evaluation on XML tree documents can be modeled as an answer set calculus, with respect to (*wrt*, for short) a set of first-order Horn clauses with constraints, in a manner that can take into account access control policies enforced on the documents. Our calculus is more general than several other access control specification formalisms, in the sense that they can all be encoded as special cases of our clausal view. The evaluation of any given XPath query Q on any given document t is formulated as the set of nodes (and the data stored there) that get selected under the runs a deterministic transition system S_Q, associated naturally with Q; the runs of S_Q on t are formulated in terms of a set of clauses (possibly with constraints) modeling the given query Q, as well as the access control policy given on t.

The paper is structured as follows: In Section 2 we briefly recall the needed notions of XPath, and fix our notation. We shall assume the queries to be *positive*, to simplify, in the sense that no negation is allowed on the navigational axes (but is allowed on the data filter parts of the query). Any such XPath query Q will be seen as a concatenation of its location steps of certain types; to each of which is associated a transition system, and the transition system S_Q for Q will be defined as the join of all these individual systems (Section 3). The evaluation of Q on any given document t is obtained under *step-wise* runs of S_Q on t; these are defined in such a way that one derives a linear complexity bound for query evaluation, wrt the size of Q (the total number of location steps composing Q), and the number of edges on t. In Section 4, we show how to integrate access control policies specified on any given document t, into our clause-based evaluation mechanism; the conditions of the policies are themselves expressed as first-order clauses over the attribute names and/or attribute values. An example is given to illustrate our approach; it underlines the necessity for keeping track of the data (on t) retrieved, by any given user, by launching a sequence of queries. Section 5 gives a few links to some of the related works; we also indicate, briefly, how our step-wise evaluation mechanism driven by clauses with constraints can actually serve as the basis for building a unified approach for handling other issues, such as the containment problem on queries visualized as patterns.

2 Preliminaries and Notation

We recall briefly some details on the XPath language (cf. [13]). The following XPath axes – referred to as *basic* in the sequel – will be used for navigation on any XML tree: `self`, `child`, `parent`, `ancestor`, `descendant`, `following-sibling`, `preceding-sibling`. (The other axes of XPath can be described in terms of these basic axes, up to a notion of equivalence; cf. e.g., [5].)

An alphabet Σ is assumed given for naming the nodes (or tags) on the documents; its elements will be referred to as nodenames or tagnames. *Att* will stand for the set of all attribute names at the nodes of all possible documents; and $att, att_1, \ldots$ will stand for variables running over *Att*. All data (including PCDATA) at the nodes on the documents will be assumed given in the form att = 'val'' where val stands for a value assignable to att.

A *filter type* is an expression generated by the non-terminal F in the following grammar – where `A` stands for a basic XPath axis among the seven mentioned above, and $\sigma \in \Sigma \cup \{*\}$ ('$*$' stands here for an 'arbitrary' element of Σ), and op is an operator in the set $Op = \{=, \neq, >, <, \geq, \leq\}$:

$L ::= @\,att\ op\ \text{'}val'\ \mid\ position() = i\ \mid\ true$
$S ::= \mathtt{A::}\sigma\ \mid\ \mathtt{A::}\sigma\mathtt{[}L\mathtt{]}\ \mid\ \mathtt{A::}\sigma\mathtt{[}L\mathtt{][}S\mathtt{]}$
$G ::= S\ \mid \mathtt{A::}\sigma\mathtt{[}L\mathtt{][}G\mathtt{]}\ \mid\ G\ or\ G\ \mid\ G\ and\ G$
$F ::= L\ \mid\ G$

The expressions of the form $\mathtt{[}F\mathtt{]}$ where F is a filter type, are said to be *filter expressions* or just *filters*. A filter expression of the form $\mathtt{[}L\mathtt{]}$, with no navigational axes, will be said to be *local*. We shall identify $\mathtt{A::}\sigma\mathtt{[}true\mathtt{]}$ with $\mathtt{A::}\sigma$. Given a context node u on any XML document t, any given filter expression $\mathtt{[}F\mathtt{]}$ evaluates either to 'true' or to 'false' at u on t; cf. e.g., [13].

Positive XPath *query expressions* are defined as the expressions Q generated by the grammar below – where `A`, σ are as above, and $\mathtt{[}F\mathtt{]}$ is any filter expression:

$Q_0 ::= \mathtt{/A::}\sigma\ \mid\ \mathtt{/A::}\sigma\mathtt{[}F\mathtt{]}\ \mid Q_0\,Q_0$
$Q' ::= Q_0\ \mid Q_0\mathtt{/attribute::}att\ \mid\ Q_0\mathtt{/attribute::}*$
$Q\ ::= Q'\ \mid\ Q'\ or\ Q'$

(Note: `attribute` is a non-navigational axis of XPath, and $/@\,att$ is short for the syntax $\mathtt{/attribute::}att$; the '$*$' in the last production stands for 'any'.) The query expressions generated by this grammar are either of the form $/C_1/\ldots/C_n$, – each C_i being of the type $\mathtt{A::}\sigma\mathtt{[}F\mathtt{]}$ –, or a disjunction of such queries; they will be referred to as *canonical*. All our queries in the sequel will be assumed to be canonical. The filter expressions in the C_i are referred to as their *filter components*. A query of the form $\mathtt{/axis::}\sigma\mathtt{[}F\mathtt{]}$ will be said to be *elementary*. A *location expression* (resp. *location step*) is an expression of the form $\mathtt{[axis::}\sigma\mathtt{[}F\mathtt{]]}$

(resp. `axis::`σ`[`F`]`) where `axis` is one of the seven (basic) navigational axes mentioned above, $\sigma \in \Sigma$, and `[`F`]` is a filter. If the filter is local, the location expression (resp. location step) will be said to be *atomic*. Any location step will be written in the form `axis::`σ`[`L`][`F`]`, where `[`L`]` is local, and the navigational axes are all in `[`F`]`.

The notion of selection of any given node v, on a document t, wrt the *Root* node of t, by any given query expression Q, is defined inductively; cf. e.g., [13]. (*Root* is the fictive root node assigned in XML to any document t, just above the actual root node of the tree t; we shall denote this latter as *root*.) For any node u on a document t, we denote by $Data_t(u)$ the data stored at u on t, and set $t(u)$ to be the pair $(\sigma, Data_t(u))$, where σ is the tagname of t at u. If Q =`/`C_1`/`...`/`C_n is any query in canonical form, the *answer* for Q on t is the set $Eval_t(Q)$ of all $t(u)|_Q$, where u is any node selected by Q on t, and $t(u)|_Q$ is the projection of $t(u)$ on the set of attributes selected by C_n.

Remark. a) We shall also need two further navigational axes that are only implicitly defined in XPath, namely the right-sibling and left-sibling axes; we shall refer to these as `right,left`, respectively; by definition, we have (cf. [13]):
`right::`σ is equivalent to `following-sibling::`$*$`[`$position() = 1$`][self::`σ`]`,
`left::`σ is equivalent to `preceding-sibling::`$*$`[`$position() = 1$`][self::`σ`]`.

b) To each `axis` among the seven basic axes above, we associate a step-wise axis denoted as `dir-axis`, the role of which is to move from node to node (on any tree t), along the direction given by `axis`. `self` is its own step-wise axis; here is the correspondence table for the other axes:

`axis`	`dir-axis`
`parent`	`parent`
`child`	`child`
`descendant`	`child`

`axis`	`dir-axis`
`ancestor`	`parent`
`following-sibling`	`right`
`preceding-sibling`	`left`

c) To any given basic XPath axis `axis`, we associate a (Boolean valued) unary predicate `Fin-axis()`, which evaluates to 'true' at any node u on any given XML document t, iff there exists no node v such that u `dir-axis` v holds on t. (We define `Fin-self()` to be always 'true'.) For an `axis` $\neq$ `self` it evaluates to 'true' at the *root*, or at the leafs, or at the right-most or left-most nodes on t.

3 The Transition System for Evaluating a Query

Let Q =`/`C_1`/`C_2`/`...`/`C_n be a canonical query; each `/`C_i is thus of the form `/`C_i =`/axis`$_i$`::`σ_i`[`L_i`][`F_i`]`, $1 \leq i \leq n$, with a local filter `[`L_i`]`, and a filter `[`F_i`]` containing all the navigational part. For redactional simplicity, we assume that the filter components of Q are free from conjunction and disjunction.

Case of an atomic elementary query: We first consider the case where `/`C_i is atomic, i.e., of the form `/`C_i =`/axis`$_i$`::`σ_i`[`L_i`]`, with a local filter `[`L_i`]`.

The elementary transition system (ETS) for the atomic elementary query C_i =`/axis`$_i$`::`σ_i`[`L_i`]`, is defined as the system S_i whose set of states is $States_i = \{init_i, ok_i, fail_i\}$, and whose transitions are defined by the following clauses

$t1, \ldots, t6$ – where $\alpha_i = \sigma_i\,\texttt{[}L_i\texttt{]}$, and $\overline{\alpha_i} = \overline{\sigma_i\,\texttt{[}L_i\texttt{]}}$ stands for the complement of the data represented by $\sigma_i\,\texttt{[}L_i\texttt{]}$:
If $\texttt{axis}_i \in \{\texttt{self, child, parent}\}$:

$t1.$ $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models \alpha_i)$, $(u\,\texttt{dir-axis}_i\,v)$;
$t2.$ $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models \overline{\alpha_i})$, $(u\,\texttt{dir-axis}_i\,v)$;

If $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$:

$t3.$ $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models \alpha_i)$, $(u\,\texttt{dir-axis}_i\,v)$;
$t4.$ $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, if $(t(v) \models \overline{\alpha_i})$, $(u\,\texttt{dir-axis}_i\,v)$;
$t5.$ $\langle fail_i, v\rangle \leftarrow \langle fail_i, u\rangle$, if $(t(v) \models \overline{\alpha_i})$, $(u\,\texttt{dir-axis}_i\,v)$;
$t6.$ $\langle ok_i, v\rangle \leftarrow \langle fail_i, u\rangle$, if $(t(v) \models \alpha_i)$, $(u\,\texttt{dir-axis}_i\,v)$;

The role of the system S_i is to help select the nodes on any given document t answering the part $\texttt{/}C_1\texttt{/}\ldots\texttt{/}C_i$ of the given query Q; this is done with the help of a run of S_i on t. The *run* of S_i on t is defined as a mapping $M_i\colon Nodes(t) \to \mathcal{P}(States_i)$, satisfying the above transition rules $t1 - t6$, plus the two additional rules below (where u, v are nodes on t, and $\langle q, u\rangle$ stands for $q \in M_i(u)$):

$1a.$ $\langle init_i, u\rangle \leftarrow u = Root$, if $i = 1$;
$\langle init_i, u\rangle \leftarrow \langle ok_{i-1}, u\rangle$, if $i > 1$;
$2a.$ $\langle init_i, u\rangle \leftarrow \langle ok_i, u\rangle$, if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$.

Semantics: The set of states $M_i(u)$ assigned to the node u is constructed incrementally under a step-wise traversal of t, and inductively wrt i. The construction starts by adding the state $init_i$ to the sets $M_i(u)$; here u stands for any node selected by the previous elementary query $\texttt{/}C_{i-1}$ for $i > 1$, and $u = Root$ for $i = 1$ (transition $1a$). The construction of $M_i(u)$ then continues – moving in the sense defined by $\texttt{axis}_i$ – by using the clauses $t1, \ldots, t6$, which allow to select the nodes where the data is consistent with the test data α_i; for any such selected node u, the state ok_i is added to $M_i(u)$; and if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$, the transition $2a$ is used for continuing the search for other nodes satisfying α_i from an already selected node. The runs of the transition system S_i are deterministic: when moving from a node u to a node v satisfying $u\,\texttt{dir-axis}_i\,v$, there exists *only one* applicable transition; as a consequence, the construction of the sets $M_i(u)$ is also deterministic in the following sense: when moving from a node u – where we have added to $M_i(u)$ some state q – to a node v satisfying $u\,\texttt{dir-axis}_i\,v$, the state to be added to $M_i(v)$ is determined unambiguously: it is ok_i if $t(v) \models \alpha_i$, and $fail_i$ otherwise.

Case of a non-atomic elementary query: We consider now an elementary query $\texttt{/}C_i$ $=\texttt{/axis}_i^0\texttt{::}\sigma_i^0\texttt{[}L_i^0\texttt{][}F_i\texttt{]}$; it can be written under the form $\texttt{/}C_i = step_i^0\texttt{[}step_i^1\texttt{[}step_i^2\texttt{[}\ldots\texttt{[}step_i^{k(i)}\texttt{]}\ldots\texttt{]}$, for some positive integer $k(i)$, where each $step_i^p = \texttt{axis}_i^p\texttt{::}\sigma_i^p\texttt{[}L_i^p\texttt{]}$ is atomic, for $0 \leq p \leq k(i)$. The transition system S_i for such an elementary subquery $\texttt{/}C_i$ is defined as the 'concatenation' of *one-step transition systems* (STS, for short) S_i^p corresponding to each of the $step_i^p$. For every given $p \in \{0, \ldots, k(i)\}$, let α_{ip} stand for $\sigma_i^p\,\texttt{[}L_i^p\texttt{]}$, and $\overline{\alpha_{ip}}$ for $\overline{\sigma_i^p\,\texttt{[}L_i^p\texttt{]}}$.

Depending on the role played by the location step $step_i^p = \texttt{axis}_i^p\texttt{::}\sigma_i^p\texttt{[}L_i^p\texttt{]}$, we have three different types of STS; in each case, the transitions are similar to those of the atomic case; only the state sets will be different (cf. [5] for full details):

- The STS S_i^0 corresponds to $\texttt{/}step_i^0\texttt{[}$ (the navigational part followed by a non-local filter). In this case, the states are $init_i^0, fail_i^0, ok_i^0[-]$; the transitions are obtained from clauses $t1 - t6$ above, by replacing $init_i, fail_i, ok_i$ respectively by $init_i^0, fail_i^0, ok_i^0[-]$.
- If $p \in \{1, \ldots, k(i) - 1\}$, the STS S_i^p, corresponds to $\texttt{[}step_i^p\texttt{[}$ (a non-local filter followed by another non-local filter). Its set of states is composed of $init_i^p[-], fail_i^p[-], fail_i^p[\bot], ok_i^p[-]$, and the transitions are obtained from the clauses $t1-t2$, by replacing $init_i, fail_i, ok_i$ respectively by $init_i^p[-]$, $fail_i^p[\bot]$, $ok_i^p[-]$, and from clauses $t3 - t6$ by replacing $init_i$, $fail_i$, ok_i by $init_i^p[-]$, $fail_i^p[-]$, $ok_i^p[-]$.
- The STS $S_i^{k(i)}$ corresponds to the last non-local filter $\texttt{[}step_i^{k(i)}\texttt{]}$ of $\texttt{/}C_i$. Its states are $init_i^{k(i)}[-], fail_i^{k(i)}[-], fail_i^{k(i)}[\bot], ok_i^{k(i)}[\top]$, and the transitions are obtained from clauses $t1 - t2$, by replacing $init_i, fail_i, ok_i$ respectively by $init_i^{k(i)}[-], fail_i^{k(i)}[\bot], ok_i^{k(i)}[\top]$, and from clauses $t3 - t6$, by replacing $init_i, fail_i, ok_i$ respectively by $init_i^{k(i)}[-], fail_i^{k(i)}[-], ok_i^{k(i)}[\top]$.

The role of S_i^0 is to find the nodes v *potentially* selected by $\texttt{/}C_1\texttt{/}\ldots\texttt{/}C_i$ (i.e., nodes v such that $t(v) \models \sigma_i^0\texttt{[}L_i^0\texttt{]}$, and such that $u\,\texttt{dir-axis}_i^0\,v$ holds for some node u that got selected by $\texttt{/}C_1\texttt{/}\ldots\texttt{/}C_{i-1}$); the systems S_i^p, for $1 \le p \le k(i)$, serve to *validate* such potential selections: among the nodes potentially selected by S_i^0, retain only those where $\texttt{[}F_i\texttt{]}$ evaluates to 'true'.

Having constructed the STSs S_i^p for $0 \le p \le k(i)$, we can now define the transition system S_i for the elementary query with a non-local filter component $\texttt{/}C_i = \texttt{/}step_i^0\texttt{[}step_i^1\texttt{[}step_i^2\texttt{[}\ldots\texttt{[}step_i^{k(i)}\texttt{]}\ldots\texttt{]}$. Its set of states is defined as:

$$States_i = \{init_i^0, fail_i^0\} \bigcup_{p=0}^{k(i)-1}\{ok_i^p[\gamma] \mid \gamma \in \{-,\bot,\top\}\}$$
$$\bigcup_{p=1}^{k(i)}\{init_i^p[\gamma], fail_i^p[\gamma] \mid \gamma \in \{-,\bot,\top\}\} \cup \{ok_i^{k(i)}[\top]\},$$

and the set of transitions is the union of the transitions from S_i^p, for $0 \le p \le k(i)$.

For any given document t, the *run* of S_i on t, is a mapping $M_i\colon Nodes(t) \to \mathcal{P}(States_i)$, consistent with the transitions of S_i, and the transitions $1 - 11$ formulated below in terms of clauses. We first introduce some notation used in these transitions: we consider a 3-valued logic over the truth table $\{1, 0, \omega\}$ with $0 < \omega < 1$; the ω here stands for 'undefined', and we set: $\overline{1} = 0, \overline{0} = 1, \overline{\omega} = \omega$. For every $p \in \{0, \ldots, k(i)\}$, we define a (3-valued) unary predicate Ω_i^p, which will be evaluated recursively, at any node u of any given document t, as follows:

i) Case where $\texttt{Fin-axis}_i^p(u)$ is true:
$\Omega_i^p(u) = 1$ iff a state of the form $q_i^p[\top]$ is in $M_i(u)$;
$\Omega_i^p(u) = 0$ iff $M_i(u)$ has a state of the form $q_i^p[\bot]$;
$\Omega_i^p(u) = \omega$ otherwise.

ii) Case where $\texttt{Fin-axis}_i^p(u)$ is false:
$\Omega_i^p(u) = Sup_v\{\Omega_i^p(v) \mid u\ \texttt{dir-axis}_i^p\ v\}$

Transition rules for M_i (u, v are nodes on t, $p \in \{0, \ldots, k(i)\}$, $\langle s, u \rangle$ stands for $s \in M_i(u)$, $q, q' \in \{init, fail, ok\}$):

1. $\langle init_i^0, u \rangle \leftarrow u = Root$, if $i = 1$;
   $\langle init_i^0, u \rangle \leftarrow \langle ok_{i-1}, u \rangle$, if $i > 1$ (rule to go from S_{i-1} to S_i)
2. $\langle init_i^{p+1}[-], u \rangle \leftarrow \langle ok_i^p[-], u \rangle$, if $p \leq k(i) - 1$ (rule to go from S_i^p to S_i^{p+1})
3. $\langle q_i^p[\top], u \rangle \leftarrow \langle q_i^p[-], u \rangle$, $\Omega_i^p(u)$, for $p \geq 1$
   (back-propagate $[\top]$ along path traversed)
4. $\langle ok_i^{p-1}[\top], u \rangle \leftarrow \langle ok_i^{p-1}[-], u \rangle$, $\langle init_i^p[\top], u \rangle$ (signal filter 'true' to $step_i^{p-1}$)
5. $\langle ok_i, u \rangle \leftarrow \langle ok_i^0[\top], u \rangle$ (validate potential selection at u)
6. $\langle init_i^0, u \rangle \leftarrow \langle ok_i, u \rangle$, (continue with M_i from a validated node u)
   if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$
7. $\langle fail_i^p[\bot], u \rangle \leftarrow \langle fail_i^p[-], u \rangle$, $\texttt{Fin-axis}_i^p(u)$, if $p \geq 1$
   (we are at an 'end', $step_i^p$ is false)
8. $\langle q_i^p[\bot], u \rangle \leftarrow \langle q_i^p[-], u \rangle$, $\overline{\Omega_i^p(u)}$, if $p \geq 1$
   (back-propagate $[\bot]$ along path traversed)
9. $\langle ok_i^{p-1}[\bot], u \rangle \leftarrow \langle ok_i^{p-1}[-], u \rangle$, $\langle init_i^p[\bot], u \rangle$, (signal filter 'false' to $step_i^{p-1}$)
   if $\texttt{axis}_i^{p-1} \in \{\texttt{self, child, parent}\}$
10. $\langle init_i^{p-1}[-], u \rangle \leftarrow \langle ok_i^{p-1}[-], u \rangle$, $\langle init_i^p[\bot], u \rangle$,
    if $\texttt{axis}_i^{p-1} \notin \{\texttt{self, child, parent}\}$
    (filter false at u for $step_i^p$, continue with $step_i^{p-1}$)
11. $\langle init_i^0, u \rangle \leftarrow \langle ok_i^0[\bot], u \rangle$, (continue with M_i from an invalidated node u)
    if $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$

Semantics: Note first that, here again, the transition system is deterministic. The role of the run M_i of S_i, on t, is to select every node u of t where $\sigma_i^0\texttt{[}L_i^0\texttt{]}$ is valid under the data $t(u)$, and the filter component $\texttt{[}F_i\texttt{]}$ evaluates to 'true'. The run M_i starts by assigning the state $init_i^0$, to the nodes v_{i-1} answering the subquery $\texttt{/}C_1\texttt{/}\ldots\texttt{/}C_{i-1}$ (clause 1). Next, using the transitions $t1 - t6$ of S_i^0, it progresses along the path defined by $\texttt{axis}_i^0$ and assigns the state $fail_i^0$ to the nodes where the data does not validate $\sigma_i^0\texttt{[}L_i^0\texttt{]}$, and the state $ok_i^0[-]$ to the first node v_i^0, such that $\sigma_i^0\texttt{[}L_i^0\texttt{]}$ is valid under $t(v_i^0)$. Every such node v_i^0 is potentially selected; it will get ultimately selected by M_i iff the filter component $\texttt{[}F_i\texttt{]}$ evaluates to 'true' at v_i^0. To check the satisfaction of the filter at v_i^0, the transitions of S_i^p, for $1 \leq p \leq k(i)$ are used. For any $0 \leq p \leq k(i) - 1$, let v_i^p denote the node to which M_i has assigned the state $ok_i^p[-]$. In this case, state $init_i^{p+1}$ will be assigned to v_i^p using clause 2. The run M_i will continue along the path defined by the $\texttt{axis}_i^{p+1}$, under the transitions of S_i^{p+1}, assigning:

- $fail_i^{p+1}[-]$ to every node where the data does not validate $\sigma_i^{p+1}\texttt{[}L_i^{p+1}\texttt{]}$,
- if $p + 1 < k(i)$, the state $ok_i^{p+1}[-]$ to the first node where the data validates $\sigma_i^{p+1}\texttt{[}L_i^{p+1}\texttt{]}$,
- if $p + 1 = k(i)$, the state $ok_i^{k(i)}[\top]$ to the first node where the data validates $\sigma_i^{k(i)}\texttt{[}L_i^{k(i)}\texttt{]}$.

The filter `[`step_i^{p+1}`]` is not satisfied at v_i^p iff at *every* w such that v_i^p $\texttt{axis}_i^{p+1}$ w, the data does not validate σ_i^{p+1}`[`L_i^{p+1}`]`. In such a case, and if $\texttt{axis}_i^p \notin \{$`self, child, parent`$\}$, the clauses $7, 8$ and 10 (with $p := p + 1$) will be used and $\mathit{init}_i^p[-]$ will get assigned to v_i^p, in order that the run M_i can continue its search for other possible nodes where the data validates σ_i^p`[`L_i^p`]`.

Validate potential selection at v_i^0: If the state $ok_i^{k(i)}[\top]$ is reached under M_i, the rules 3, 4 are used to propagate the validation symbol $[\top]$ at any node u traversed by M_i. In particular, $ok_i^0[\top]$ is added to $M_i(v_i^0)$; and by rule 5 the selecting state ok_i is also added to $M_i(v_i^0)$.

Invalidate potential selection at v_i^0: The potential selection at v_i^0 is invalidated if the filter `[`F_i`]` evaluates to 'false' at v_i^0. In this case, we propagate the invalidation symbol $[\bot]$ by using the rules 7–9. In particular, the state $ok_i^0[\bot]$ is added to $M_i(v_i^0)$; such a node will not get selected.

Rules 6 and 11 serve to continue the run M_i from a validated or invalidated potentially selected node v_i^0.

The transition system for the full query: The transition system S_Q, for the full query $Q =$`/`C_1`/`C_2`/`$\ldots$`/`C_n, is defined as the system whose set of states is $\mathit{States}_Q = \bigcup_{i=1}^n \mathit{States}_i$, and the set of transitions is composed of the transitions of the systems S_i, for $1 \leq i \leq n$. A run M_Q of S_Q, on any given document t, is defined as a mapping $M_Q\colon \mathit{Nodes}(t) \to \mathcal{P}(\mathit{States}_Q)$, with $M_Q = \bigcup_{i=1}^n M_i$, with the following two additional transitions (whose semantics must now be evident):

12. $\langle \mathit{init}_1, u\rangle \leftarrow \langle ok_n, u\rangle$, if $\texttt{axis}_1 \notin \{$`self, child, parent`$\}$
13. $\langle \mathit{init}_1^0, u\rangle \leftarrow \langle ok_n, u\rangle$, if $\texttt{axis}_1^0 \notin \{$`self, child, parent`$\}$

Just like for ETSs or STSs, a run M_Q of the system S_Q (for the query Q on the document t) constructs the set $M_Q(u) \subset \mathit{States}_Q$ at any node u incrementally, in a step-wise fashion; M_Q starts at the *Root* and traverses t along the various axes of the successive location steps in Q. The answer for Q on t is the set of all nodes u of t, such that $ok_n \in M_Q(u)$, at the end of the run.

A Linear Optimizing Strategy: By a transition, or "move", of S_Q from any given node u, we shall mean a resolution step from a positive clause $\langle q, u\rangle \leftarrow$, into one of the clauses $t1$-$t6$, $1a$-$2a$ or 1-13 given above, defining M_Q. Note that such a 'move' of S_Q may not always be from u to a different node v on t: it may just add a further state to the set $M_Q(u)$, at the current node u.

The run M_Q can actually be controlled with a strategy – referred to as the *linear strategy*– to avoid redundant transitions, and reduce the complexity of the evaluating run of S_Q. It is based on the simple idea that at any node u, the satisfaction of step_i^p (resp. step_i in atomic case) need not be checked more than once; this strategy is formalized in the proof of the following proposition:

Proposition 1. *The complexity of the step-wise evaluation of any XPath query Q on any XML document t by the run of S_Q is linear on the number of atomic location steps in Q and the number of edges on t.*

Proof. We stick to the notation above. For every $i, p, 1 \leq i \leq n, 0 \leq p \leq k(i)$, and for any given edge (u, v) on the document t, we associate a set $S_i^p(u, v)$; initially – at the start of the evaluating run M_Q of S_Q on t – this set is composed of all the transitions of the system S_i^p; whenever the run M_Q moves from a node u to a node v, using one of the transition rules, say ρ, of the system S_i^p, we remove *all* the transition rules of S_i^p having the same head literal as ρ from the set $S_i^p(u', v)$, for *every* node u'. Such an evolution of the sets $S_i^p(u, v)$, under the moves of M_Q, is justified due to the fact that S_Q is deterministic. Any subsequent move under the run M_Q is then allowed to use only the transitions still present in $S_i^p(u, v)$. (Note: transitions could be from a node to itself, so we shall agree that t has "fictive" edges from any node u to itself, for the purposes of this strategy.)

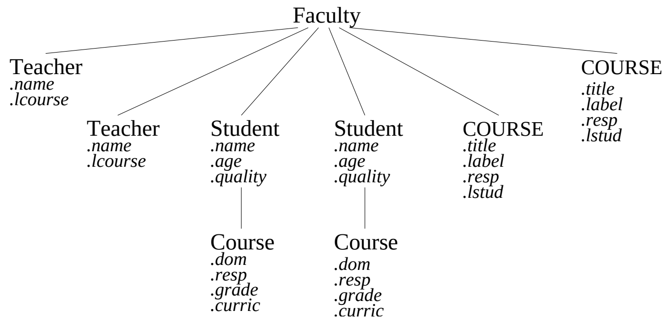
In particular, under the evaluating run M_Q, no edge on t is traversed by using more than once any of the transitions of the system S_Q. □

An example illustrating the evaluation of a query under a run with the linear strategy is given in *Appendix-I.*

4 Access Control

Various methods have been proposed for controlling the access to data (cf. e.g., [7]). The approach of [10] allots access keys to nodes, or more generally to subsets of attributes at the nodes; the keys could depend on a notion of category to which the consultant belongs. In [3] is suggested a somewhat different approach: to every category of consultants, associate a set of first-order clauses with constraints. [12] proposes to attach some special labels to the nodes of the document, such as $-r, -R$, to signify 'local' (resp. 'recursive') access denial. Our proposal in this paper is to model access control policies as first-order clauses. (It is shown in *Appendix-II* that such a view can cover all the approaches mentioned above.) The clauses modeling access control will be subject, in general, to some constraints referred to as *scope constraints*, as illustrated in in the following example:

Example 1. Suppose given an (XML-format) database "Faculty", with as children nodes "Teacher", "Student", and "COURSE".


Faculty
Teacher .name .lcourse
Teacher .name .lcourse
Student .name .age .quality
Student .name .age .quality
COURSE .title .label .resp .lstud
COURSE .title .label .resp .lstud
Course .dom .resp .grade .curric
Course .dom .resp .grade .curric

Here is an access control policy on this document, formulated in plain text, for two categories of users consulting such a document: *Adm* (resp. *Acad*) for administrative (resp. academic) staff.

Adm: a user of this category may not have access both to the name of a student and to the grade obtained in any of the course the student follows; but access to either one of these two data is allowed.

Acad: a user of this category is allowed access to the name of a student, and to the grade obtained by the student in any course, *iff* the user is the person giving the course.

We consider any category as a unary predicate, evaluating to 'true' only on the identifiers assigned to each user of the category. The policy for *Adm* can then be modeled as the following pure negative clause:

(1): $\leftarrow Adm(*)$, Student.name, Course.grade,
[Student.name `child` Course.grade]

where '$*$' stands for any identifier for a user of the category *Adm*. The intended meaning is that, for a user of category *Adm*, the policy is violated if the knowledge (s)he gains by consulting the database (possibly repeatedly), contains the name of a student, as well as the grade obtained by the student in a course; the scope constraint (in square brackets) says that the attributes 'name' and 'grade' are at two `child`-related nodes with tagnames 'Student' and 'Course', respectively.

The current state of data, stored at a node v, disclosed directly or indirectly to the user with identifier *id*, having launched a query Q (and possibly also some other queries prior to Q), will be represented by a set of pure positive clauses, containing at least the positive clause $Cat(id) \leftarrow$, where Cat stands for the category of the user. Suppose we are at a context node u on t, under the run of the transition system S_Q, and a node v is to be reached under the next transition of S_Q; this transition will correspond to a well-determined location step Q' of Q; a set $Deduce_{id}(v, Q')$ of positive clauses will then be constructed to model the knowledge that the user can acquire – directly or indirectly – on the data stored at v on t, by firing this transition. By definition, then, the run of S_Q on t will select the node v, if and only if:

$$Deduce_{id}(v, Q') \cup \{(1)\} \not\models \perp$$

where (1) is the negative clause above, modeling the access control policy for the category *Adm*. The positive clauses of $Deduce_{id}(v, Q')$ will all be added to the history record of the user, for controlling the data (s)he accesses under future queries. The construction of the sets of clauses $Deduce_{id}()$ needed for such a clause-based, step-wise evaluation of queries, is done below.

Definitions: We formalize here the clause-based view for access control.

Definition (1): By *User*, we mean a (finite) set of individual users; and by *Category* we shall mean a finite set of users, with a given specific name or label (as in the Example above). Both are seen as unary predicates: *User()*, *Cat()*. Consider a given document t, on which a given access control policy has been enforced; it will be assumed that every access condition of the policy is expressed as a pure negative clause, possibly subject to a scope constraint (as in the Example above); the set of all such negative (constrained) clauses will be denoted as $\mathcal{P}$. Every literal appearing in any of these clauses is therefore either a unary predicate (of the form *User()* or *Cat()*), or a propositional symbol of the form

$u.a$ where u is a nodename, and a is an attribute name. The *scope constraint* of a clause in $\mathcal{P}$ is, by definition, a constraint of the form $[constr]$ where *constr* is a (finite) conjunction of expressions of the form $u_i.a_i\ \texttt{axis}_i\ v_i.b_i$ (where $\texttt{axis}_i$ is a basic axis or a corresponding `dir-axis`), and/or of expressions of the form $u_j.a_j\ op\ val$, where *val* is a data value, and $op \in Op$.

For instance, a policy clause of the form:

$\leftarrow$ *User*(10), u.a, v.b, w.c, [u.a `child` v.b, u.a `right` w.c]

stipulates that a user with identity 10 is not allowed access to the data stored by attribute a at node u, by attribute b at node v, and by attribute c at node w, all three together; by the "all three together", it is meant that such a combined access is disallowed to *User*(10), *under one or more queries* on the document.

Definition (2): At any node u of t, if a set $\mathcal{D}_u(t)$ of defining functional relations is specified on the data stored at u on t, then we also assume that $\mathcal{D}_u(t)$ is formulated as a set of Horn clauses of the form $u.a_{i_r} \leftarrow u.a_{i_1}, u.a_{i_2}, \ldots, u.a_{i_k}$, where the a_j are attribute names at u.

The document t being given, we shall drop references to t as indices (or otherwise), in the definitions below.

Definition (3): For any query $Q = /C_1/C_2/\ldots/C_n$ that is being evaluated by a run of its associated transition system S_Q on the given document t, suppose a transition of the run, from a context node u to a node v on t, is a transition of a one-step transition system S_k^p for some p, k; then, by the *current query-step* of Q at u, we mean the corresponding location step $step_k^p$ in Q (notation of the previous section).

Definition (4): For any given query Q launched by a given user with identity *id*, and a run of the associated system S_Q, we define three sets of positive clauses – denoted as $Hist_{id}(v, Q)$, $Scope_{id}(v, Q)$ and $Access_{id}(v, Q)$ – at every node v traversed by the run; these three sets are constructed inductively, in the following manner, wrt the the various queries successively launched by the user. Let $Q^{(0)}, Q^{(1)}, \ldots, Q^{(i)}, \ldots$ denote the sequence of the successive queries launched by the given user; the identity *id* of the user once fixed, we omit it, for readability, in the constructions below (where '*root*' is the actual root node of the tree t):

Step 0. Set $i = 0$.

Step 1. Case v is the root node of t: Define $Scope(root, Q^{(i)}) = \emptyset$;
if $i = 0$, then $Hist(root, Q^{(i)})$ is a singleton: $\{User(id) \leftarrow\}$ or $\{Cat(id) \leftarrow\}$;
if $i > 0$, $Hist(root, Q^{(i)})$ is defined as $Hist(root, Q^{(i-1)})$;
if $i = 0$, then $Access(root, Q^{(i)})$ is set to be the emptyset;
if $i > 0$, $Access(root, Q^{(i)})$ is defined as $Hist(root, Q^{(i-1)})$.

Step 2. Case v is a non-root node:
Let u be the context node for the current transition of $S_{Q^{(i)}}$ to the node v. For all nodes u' traversed prior to u (including u), by the run of $S_{Q^{(i)}}$ evaluating $Q^{(i)}$ (i.e., the i-th query launched by the given user), assume having constructed the sets $Access(u', Q^{(i)})$, $Scope(u', Q^{(i)})$, and $Hist(u', Q^{(i)})$. The sets $Scope(v, Q^{(i)})$, $Access(v, Q^{(i)})$, and $Hist(v, Q^{(i)})$ are then constructed as follows:

1. Let `axis` be the axis of the current query-step at u, and suppose there is a node u' on t already traversed by the run such that $u'\ \texttt{axis}\ v$ holds on t;

if a (resp. att) is an attribute name at u' (resp. at v), such that $u'.a\,\texttt{axis}\;v.att$ appears in the scope constraint of some policy clause, then create a positive 'scope clause' $[u'.a\,\texttt{axis}\,v.att] \leftarrow$; and define $Scope(v, Q^{(i)})$ as $Scope(v, Q^{(i-1)}) \cup \{[u'.a\,\texttt{axis}\,v.att] \leftarrow\}$

2. For every $v.att$ appearing in the body of a policy clause in $\mathcal{P}$, such that $v.att$ is consistent with $\sigma\texttt{[}L\texttt{]}$ (where the current query-step is of the form $\texttt{axis::}\sigma\texttt{[}L\texttt{]}$), create a positive 'data-access clause' $v.att \leftarrow$, and set
$$Access(v, Q^{(i)}) = Access(v, Q^{(i-1)}) \cup \{v.att \leftarrow\}.$$
3. Set $Hist(v, Q^{(i)}) = Hist(v, Q^{(i-1)}) \cup Scope(v, Q^{(i)}) \cup Access(v, Q^{(i)})$.

Step 3. Set $i = i + 1$, and GOTO **Step 1**.

Definition (5): In the presence of an access control policy $\mathcal{P}$ on t, a transition of the system S_Q, from a context node u to some node v on t, assigns a selecting state to the node v if and only if: $Hist(v, Q^{(i)}) \cup \mathcal{D}_v(t) \cup \mathcal{P} \not\models \bot$.

Effectively Using the Method: i) Clausal resolution is essential for the step-wise query evaluation technique proposed above. In addition to the usual first-order rules for resolving between positive and negative literals, we shall also need some additional rules, such as those in the following non-exhaustive list:

1. a literal $u.att$ can resolve with a literal (of the opposite sign) of the form $u.*$; the same also for literals (of opposite sign) of the form $User(id)$ and $User(*)$, etc.
2. a scope literal $[u.a\ \ \texttt{child}\ \ v.b]$ can resolve with a scope literal of the form $[v.b\ \ \texttt{parent}\ \ u.a]$, etc.
3. a negative scope literal $[u.a\ \ \texttt{axis}\ \ v.b]$ can resolve with a positive scope literal of the form $[u.a\ \ \texttt{axis}\ \ v.b\texttt{,}\ \ u.a \neq \text{`}val'\text{'}]$

ii) It is necessary to keep the history records at the nodes, if the access control policy is not to be violated. For instance, consider the policy specified for the category *Adm*, in the Example of Section 4. Let us suppose, for instance, that the *grade* data stored at every *Course* node, is not a constant value (i.e., students from different grades follow the course). Suppose then that no history records are kept (for the data accessed), and that a given user of category *Adm* launches the following three queries, successively:
$Q_1 = \texttt{//}Student\texttt{/}@*$
$Q_2 = \texttt{//}Course\texttt{/}@grade$
$Q_3 = \texttt{//}Course\texttt{[}@grade \neq \text{`}g1'\texttt{]/parent::}Student\texttt{/}@name$
where $\texttt{//}*$ is short for $\texttt{/descendant::}*$. From the first query, one gets all the data stored at every *Student* node, in particular all the names of the students; from the second query, one retrieves all the *grade* data stored at every *Course* node; from the third, the user can get the names of all students, not registered for grade $g1$; (s)he can then deduce the name(s) of all students registered for grade $g1$, by complementation, and this will violate the given access policy, for the category *Adm*. The second query has only served to deduce that there *is* a *grade* attribute with value $g1$.

Let us now analyze the same three queries Q_1, Q_2, Q_3, when history records are kept. At the end of the evaluation of Q_1 and Q_2, these records will contain the following positive clauses:

$Adm(id) \leftarrow$,
$student.* \leftarrow$,
$course.grade \leftarrow$.

During the evaluation of the query Q_3, the following positive scope clause will get added to the history:

$$[course.grade \ \ \texttt{parent} \ \ student.name, course.grade \neq `g1'] \leftarrow .$$

The semantics of such an addition implies, in particular, that the scope constraint $[course.grade \neq `g1']$ evaluates to true. The set of all positive scope and data-access clauses thus generated, history inclusive, is inconsistent with the access policy specified for the category *Adm*; this is easily checked by resolution. As a result, the third query will *not* produce an answer, i.e., will not select any $student.name$.

(Note: A query of the form: $//Course$`[`$@grade = `g1'$`]/parent::`$Student$`/` $@name$, or of the form: $//Student$`[`$@name = `n1'$`]/`$Course$`/`$@grade$, would have led to a violation of access control without any need for history, so would have produced no answer.)

iii) The step-wise evaluation method described above, is *deductively complete* in the sense of the following proposition (its proof follows from definitions):

Proposition 2. *Let $\mathcal{P}$ be a set of negative clauses, implementing an access control policy for a given category of users, on a document t. Then a piece of data, stored on t by an attribute* att *at some node with tagname u on t, is accessible to a user of this category (under* some *XPath query)* iff *$u.att \leftarrow$ is element of a set $\mathcal{S}$ of positive data-access and scope clauses, which is consistent with $\mathcal{P}$.*

5 Related Work, Perspectives and Conclusion

Access control to XML documents has already been studied by several authors; we only mention here a few where the approach is related to ours. The approach of [10, 2] is based on the attribution of keys to the nodes and/or the data on the document. In [12], labels are attributed to the nodes, to signify local or recursive accessibility or denial. In *Appendix-II* we show how to encode such approaches in terms of clauses. [3] focuses on a notion of non-interference of a given policy wrt a given query, based on type inferences; and access control via clauses is only suggested at the end. [7] is a survey presenting the generalities on logical frameworks suited for access control, but the concern is not on query evaluation techniques. In [9], the authors propose a semantic web rule language (SWRL), as an extension of the web ontology language (OWL) using Horn Clauses and the XACML language; role based access control policies (RBAC) can be specified in SWRL, in a natural manner. It is not hard to extend the clausal view that we have presented in this paper, to cover role based access control. We also hope to consider more general access control clauses of the form presented in [1], and still preserve our step-wise evaluation technique for queries.

The idea of constructing, for any given query Q, a node selecting transition system S_Q has some similarity with the automaton associated with Q in [8]; but our system S_Q is not constructed in one single piece, and it serves a very different purpose. It is not difficult to extend the result of Proposition 1 in the presence of an access control policy, to get a complexity bound for query evaluation that is linear on the number of atomic location steps in Q, the number of edges of t, *and* the size of the data stored on t (note that the size of the history record can always be bounded by the size of the data on t).

The view we have presented in this paper for query evaluation, based on the runs of suitably defined transition systems, can also be adapted for handling other issues, such as e.g., the problem of patterns and pattern containment studied in [11]. Thus, the pattern to the left of Figure 1 below represents the XPath query `/descendant::`Student/Course:

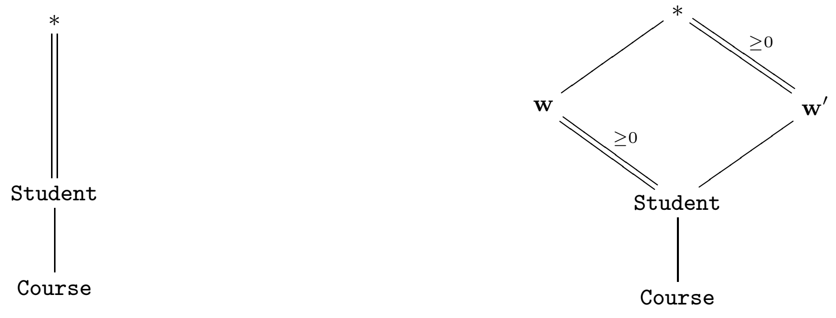


Fig. 1. Pattern, and corresponding Generalized Pattern

To the right of the figure is what we shall call the corresponding *generalized pattern* (*g-pattern*, for short); the edges $\overset{\geq 0}{=\!=\!=}$ stand for the `descendant-or-self` axis of XPath. A g-pattern can be seen naturally as a transition system, and a notion of *covering run* of this system can be defined on any given document or g-pattern. This generalizes the notion of pattern homomorphism of [11], and with such a generalized notion, the pattern containment problem can be satisfactorily handled; the details will appear elsewhere.

Access control policies can be formulated in terms of g-patterns as well: for instance, the following access control policy:

$\leftarrow$ *Adm*($*$), Student.name, Course.grade [Student.name `child` Course.grade] can be encoded as a (negative) constraint on the `child` edge at the lower level, on the g-pattern to the right of Figure 1.

It is thus possible to build a uniform approach on the basis of the clausal view we have developed in this paper, that can in particular be adapted to check if a given document t is in conformity with a given DTD; it will also be applicable if the document is given in a compressed form as a DAG, not necessarily as a tree (cf. [4]). Finally, as concerns negation on the navigational part of the queries, our hope is to handle it with suitable 'built-in' unary predicates, as in [6]; this

– as well as completing the implementation of the approach, which is currently under way – is part of our future work.

References

1. M. Abadi, *Logic in Access Control* In Proc. LICS'03, IEEE, pp. 228–233,
2. M. Abadi, B. Warinschi, *Security Analysis of Cryptographically Controlled Access to XML Documents* In Proc. of PODS'05, pp. 108–117, ACM, June 2005,
3. V. Benzaken, M. Burelle, G. Castagna, *Information Flow Security for XML Transformations*, In Proc. of ASIAN'03, pp. 33–53, LNCS 4246, Springer-Verlag, 2003,
4. B. Fila, S. Anantharaman, *Automata for Positive Core XPath Queries on Compressed Documents*, In Proc. of the 13th Int. Conf LPAR'06, pp. 467–481, LNAI 4246, Springer-Verlag, 2006,
5. B. Fila, S. Anantharaman, *A Clausal View for Access Control and XPath Query Evaluation*, Research Report available at: `www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2007/RR-2007-12.ps.gz`
6. M. Frick, M. Grohe, C. Koch, *Query Evaluation on Compressed Trees*, In Proc. of LICS'03, IEEE, pp. 188–197.
7. I. Fundulaki, M. Marx, *Specifying Access Control Policies for XML Documents with XPath* In Proc. of SACMAT'04, pp. 61–69, ACM, 2004.
8. T. Green, G. Miklau, M. Onizuka, D. Suciu, *Processing XML Streams with Deterministic Automata*, In ACM Trans. on Database Systems, 29(4):752–788, 2004.
9. H. Li, X. Zhang, H. Wu, Y. Qu *Design and Application of Rule Based Access Control Policies*, Semantic Web and Policy Workshop, Nov. 2005. at `www.csee.umbc.edu/swpw/papers/`
10. G. Miklau, D. Suciu, *Controlling access to published data using cryptography*, In Proc. of VLDB'03, pp. 898–909, 2003.
11. G. Miklau, D. Suciu, *Containment and Equivalence for a Fragment of XPath*, In Journal of the ACM, 51(1):2-45, 2004.
12. M. Murata, A. Tozawa, M. Kudo, *XML Access Control Using Static Analysis*, In Proc. of the 10th ACM Conf. on Computer and Communications Security (CCS'03), pp.73–84, ACM, 2003.
13. World Wide Web Consortium, *XML Path Language (XPath Recommendation)*, Available at: `www.w3c.org/TR/xpath/`

Appendix-I: An Evaluating run illustrated

We evaluate here the query $Q = \texttt{/descendant::}a\texttt{[ancestor::}b\texttt{]/parent::}c$, on the tree t represented in Figure 2.

The nodes of t are represented by their positions. Figure 3 presents the part of the run of the transition system S_Q, evaluating the first elementary query $/C_1 = \texttt{/descendant::}a\texttt{[ancestor::}b\texttt{]}$. We first consider the navigational part $step_1^0 = \texttt{descendant::}a$. Starting from the fictive *Root* of t, the run traverses the document top-down (`descendant` axis), using the following clauses:

$$\begin{array}{ll}
\langle init_1^0, Root\rangle \leftarrow & \langle ok_1^0[-], 2\rangle \leftarrow \langle fail_1^0, \varepsilon\rangle \\
\langle fail_1^0, \varepsilon\rangle \leftarrow \langle init_1^0, Root\rangle & \langle ok_1^0[-], 11\rangle \leftarrow \langle fail_1^0, 1\rangle \\
\langle fail_1^0, 1\rangle \leftarrow \langle fail_1^0, \varepsilon\rangle & \langle ok_1^0[-], 12\rangle \leftarrow \langle fail_1^0, 1\rangle
\end{array}$$

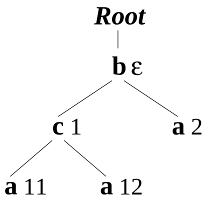


Fig. 2. XML document t

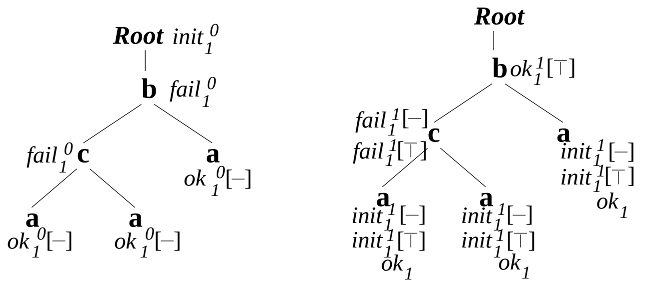


Fig. 3. Evaluation of `/descendant::`a`[ancestor::`b`]` on t

The states assigned to the nodes of t are shown in Figure 3, to the left. The run of $step_1^0$ uses the clause $\langle init_1^1[-], u\rangle \leftarrow \langle ok_1^0[-], u\rangle$, and starts its bottom-up move for evaluating the filter part $step_1^1 =$ `ancestor::`b, at the nodes $u = 2$, 11, 12, (Figure 3, to the right). The following clauses are then used for the run of the STS S_1^1:

$$\langle fail_1^1[-], 1\rangle \leftarrow \langle init_1^1[-], 11\rangle \qquad \langle fail_1^1[-], 1\rangle \leftarrow \langle init_1^1[-], 12\rangle$$
$$\langle ok_1^1[\top], \varepsilon\rangle \leftarrow \langle fail_1^1[-], 1\rangle \qquad \langle ok_1^1[\top], \varepsilon\rangle \leftarrow \langle init_1^1[-], 2\rangle$$

By the linear strategy, we avoid using the clauses $\langle fail_1^1[-], 1\rangle \leftarrow \langle init_1^1[-], 12\rangle$ and $\langle ok_1^1[\top], \varepsilon\rangle \leftarrow \langle init_1^1[-], 2\rangle$ twice, for reaching the nodes 1 and ε.

Next, using the rule 3 from the definition of the run, the symbol $[\top]$ gets back-propagated from the node ε to the nodes 1, 2, and then from 1 to 11 and 12. Finally, with the help of the rules:

$$\langle ok_1, u\rangle \leftarrow \langle ok_1^0[\top], u\rangle, \qquad \langle init_2, u\rangle \leftarrow \langle ok_1, u\rangle$$

the state ok_1 is assigned to the nodes $u = 11$, 12, 2, which become the context nodes for the second elementary query $/C_2 =$ `/parent::`c.

The evaluation of $/C_2$ is presented in Figure 4. Starting from the context nodes $11, 12$ and 2, the run moves bottom-up; and using the clauses:

$$\langle ok_2, 1\rangle \leftarrow \langle init_2, 11\rangle, \quad \langle ok_2, 1\rangle \leftarrow \langle init_2, 12\rangle, \quad \langle fail_2, \varepsilon\rangle \leftarrow \langle init_2, 2\rangle,$$

it selects the node 1, and refuses the selection of the node ε. By the linear strategy, we avoid using the clause $\langle ok_2, 1\rangle \leftarrow \langle init_2, 12\rangle$ for moving to node 1, for a second time.

Under this run, we have selected the node 1. (In the general case, we should then use the clause $\langle init_i^0, 1\rangle \leftarrow \langle ok_2, 1\rangle$, to look for the other nodes possibly answering Q.)

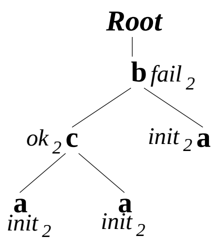


Fig. 4. Evaluation of …/`parent::`c

Appendix-II: Expressive Power of the Clausal View

We show here how to encode, in terms of clauses, the approaches for access control that we evoked in Section 5. We shall adopt the following terminology and notation: The terms *User* and *Category* will carry the same meanings as in Example 1 above. A *Key* K, on a document tree t, is defined as a (partial) function on the set $Nodes_t \cup Att$ composed of nodes on t and attribute names, such that $K(x)$ is a non-empty subset of *Users*, when defined; thus $K(x)$ is seen as a category, with some additional properties, such as propagating access restrictions to children nodes (see below); we also assume given a set with two symbols: $Acc = \{-r, -R\}$, which will play the same access denial role as in [12]. The clauses below are the translations of the access policies in the aforementioned approaches; in these clauses X stands for 'any' user in *User*, and A, B stand for specified users; u, v are nodes on any given document t; '$*$' stands for 'any' attribute at the nodes concerned; and when a key $K(x)$ is mentioned, it is assumed to be non-empty:

i) Clauses for local or recursive denial at a node:
- $\leftarrow X, u.* \, [Acc(u) = -r]$ /* local denial */
- $\leftarrow X, u.* \, [Acc(u) = -R]$ /* recursive denial */
- $\leftarrow X, v.* \, [Acc(u) = -R,\ u.* \ \texttt{child}\ v.*]$ /

ii) Clauses for key attribution to a node:
- $\leftarrow B, u.* \, [B \notin K(u)]$
- $\leftarrow B, v.* \, [B \notin K(u),\ u.* \ \texttt{child}\ v.*]$

iii) Clause for key attribution to an attribute at a node:
- $\leftarrow B, u.@att \, [B \notin K(att)]$

For the case where access control is based on the attribution of keys to the nodes and/or the data, we first observe that a query $/C_1/\ldots/C_n$ in canonical form has a non-empty answer on any t, only if the `axis` of its first elementary subquery $/C_1$ is either `child`, or `descendant`. This allows the history record for any user with some identity B, to account for a key K assigned to a node u on the document, by proceeding as follows, whatever be the query Q:

• if the access policy $\mathcal{P}$ contains a clause $\leftarrow B, u.* \, [B \notin K(u)]$, then add the positive clauses $B \leftarrow$, $u.* \leftarrow$ to the set $History_B(Q)$, and do likewise at every v below u on t, whenever there is a transition of S_Q from u to v.

A policy that assigns the symbol $-r$ or $-R$, for local or recursive access denial at the nodes, is handled similarly.