A Formal Language for XML Authorisations Based on Answer Set Programming and Temporal Interval Logic Constraints

Sean Policarpio and Yan Zhang

Intelligent Systems Laboratory
School of Computing and Mathematics
University of Western Sydney
Penrith South DC, NSW 1797, Australia
{spolicar, yan}@scm.uws.edu.au

Abstract. The Extensible Markup Language is susceptible to security breaches because it does not incorporate methods to protect the information it encodes. Our work presented in this paper focuses on the development of a formal language which can provide role-based access control to information stored in XML formatted documents. Our language has the capacity to reason whether access to an XML document should be allowed. The language, $\mathcal{A}^{xml(T)}$, allows for the specification of authorisations on XML documents and distinguishes itself from other research with the inclusion of temporal interval reasoning and the XPath query language.

Keywords: AI in computer security, logic programming, knowledge representation and reasoning, access control, authorisations, XML databases and security

1 Introduction

The Extensible Markup Language [WWW Consortium, 2008] has steadily become a common encoding format for software applications. It is a popular and reliable formatting structure for the storage, presentation, and communication of data over the Internet. Many applications use XML to encode important, and in many cases, private information. Because XML does not have an inherent security model as part of its specification there is a necessity for methods in which access to XML documents can be controlled [WWW Consortium, 2008].

In this paper, we present the development of a formal language that will provide access control to XML documents. $\mathcal{A}^{xml(T)}$ is used to define a security policy base capable of specifying all the access rights that subjects in the scope of an XML environment should have or be denied.

The formal language has particular aspects that differ from most other implementations. First, it incorporates the XML query language, XPath, into it for the purpose of defining which documents (or elements within a document) we would like to restrict access to [WWW Consortium, 1999]. An XPath is a string representation of traversing through an XML document to return an element within the document. For example, the following is an XPath that follows the tree-like structure of a document to return the element *author*:

```
/library/books/book/author
```

XPath also includes other interesting features. These include, but are not limited to, *XPath predicates* and *wildcards* which allow for broader and much more expressive XPath queries [WWW Consortium, 1999]. As opposed to *static* XPath's which are only meant to return specific nodes within XML documents, we can use these features to write *dynamic* paths that can represent zero to many elements within the database of documents.

Secondly, the formal language uses the Role-based Access Control model [Ferraiolo et al., 1995] as a basis for the structure of authorisations to subjects. This primarily means rather than applying authorisations directly to subjects, we create *roles* that can have one or more specified authorisations. This gives us better control over which subjects have what authorisations and is the foremost reason this model is chosen over others (ie. Discretionary and Mandatory Access Control models) [Ferraiolo et al., 1995]. Consequently, it also allows us to easily incorporate the principles of *separation of duty* and *conflict resolution* directly into the language [Ferraiolo et al., 1995].

Finally, we incorporate temporal interval logic reasoning into the formal language. Temporal intervals are representative of specific sections of quantitative time. Temporal interval logic is the study of relating these various points and sections of time with each other. We use temporal intervals in our formal language for the purpose of specifying *when* authorisations to XML documents should be applied. We also use temporal logic to reason upon relationships that authorisations could have with each other with respect to time.

Temporal logic is a well studied field and many models or methods have been proposed in the last decades. For our purposes, we choose to use Allen's Temporal Interval Relationship algebra [Allen, 1984]. Allen's temporal relationships cover all possible ways in which intervals can relate to one another (such as *before*, *meets*, *equal*, etc.) and are incorporated into the syntax of our formal language. However, it should be noted that what makes Allen's temporal interval logic differ from others, and what makes it appealing for our work, is that it forgoes relating intervals with specific quantities of time. Simply, Allen's logic relates intervals without the need to specify or know exactly when an interval takes place. This is possible due to the fact that *when* a temporal interval takes place is implied by its relationship(s) with all other intervals. Therefore, for an interval to exist and be relevant, it only need have at least one of Allen's relationships with at least one other interval.

The semantics of our formal language is provided through its translation into a logic program. Answer Set Programming (ASP) is a relatively new form of programming in the field of knowledge representation and reasoning. It is a form of declarative programming for search problems involving nonmonotonic reasoning and is based on Gelfond's and Lifschitz's stable model semantics of logic programming [Baral, 2003,Gelfond and Lifschitz, 1988,Lifschitz, 2008].

ASP is used to represent known information which can be reasoned upon to produce further knowledge or *answers* based on the validity of said information. This is possible because the initial information can be nondeterministically written with variableness so that different outputs can be computed from it. Simply, we can describe a scenario with an understanding that various conclusions or *answer sets* are achievable within it. We can then query under what conditions those conclusions can be met.

Access control specific to XML documents is an issue still sought out in the field of computer security. There have been different approaches to the problem. One of those approaches involves the principle of the fine-grained access control model [Damiani et al., 2002]. This model takes an XML document and designates access rights on each element. Implementations of rule propagation, positive and negative authorisations, and conflict resolution exist in the model. Through their algorithm, a source XML document can be processed by removing all objects of negative authorisation and returning a document with only elements that are allowed to be viewed [Damiani et al., 2002]. In most cases, this is the general framework for XML access control. However, we believe that an approach that closely resembles the role-based access control model is preferred.

In [Bertino et al., 2000,Bertino et al., 2004], they discussed their own implementation of an access control system for XML documents. Their work does follow the role-based access control model to a certain extent (we did not see methods for role propagation or separation of duty). Subjects are granted authorisation through credentials and objects are specified through XPath's. The implementation includes features such as the propagation of policy rules and conflict resolution. Bertino et al. include in their formalisation temporal constraints based on their previous work in [Bertino et al., 1998]. However, their approach seems restricted in terms of handling XPath expressions in authorisation reasoning.

Besides Bertino et al., only a small group of other researchers have produced research utilising logic programming for XML policy base descriptions [Anutariya et al., 2003,Gabillon, 2005]. To the best of our knowledge, a logic-based formal language for XML authorisations has not yet been developed with the inclusion of temporal constraints, the complete role-based access control model, and nonmonotonic reasoning capabilities of answer set programming.

The aim of this paper is to introduce the application and expressive power of our formal language of authorisation for XML documents. The rest of the paper is organised as follows. Section 2 presents the formal syntax of our language $\mathcal{A}^{xml(T)}$, illustrates its expressive power through various XML access control scenarios, and defines queries on XML policy bases. Section 3 describes the semantics of language $\mathcal{A}^{xml(T)}$ based on its translation into a logic program under answer set semantics. In section 4, a case study is presented to show the application of $\mathcal{A}^{xml(T)}$ in XML authorisation specification and reasoning.

```
                  <rule> ::= <head-statement> [ if [ <body-statements> ] [ with absence
                             <body-statements> ] ]
             <deny-rule> ::= admin will deny [ if [ <body-statements> ] [ with absence
                             <body-statements> ] ]
        <head-statement> ::= <relationship-statement> | <grant-statement> |
                             <query-statement> | <auth-statement> |
                             <role-statement>
       <body-statements> ::= <body-statement> | <body-statement>, <body-statements>
        <body-statement> ::= <relationship-statement> | <grant-statement> |
                             <query-statement> | <auth-statement> |
                             <role-statement>
<relationship-statement> ::= admin says <relationship-atom>
       <grant-statement> ::= admin grants <role-name> to <subject> during
                             <temporal-interval>
     <relationship-atom> ::= below( <role-name>, <role-name> ) |
                             separate( <role-name>, <role-name> ) |
                             during( <temporal-interval>, <temporal-interval> ) |
                             starts( <temporal-interval>, <temporal-interval> ) |
                             finishes( <temporal-interval>, <temporal-interval> ) |
                             before( <temporal-interval>, <temporal-interval> ) |
                             overlap( <temporal-interval>, <temporal-interval> ) |
                             meets( <temporal-interval>, <temporal-interval> ) |
                             equal( <temporal-interval>, <temporal-interval> )
               <subject> ::= <subject-constant> | <subject-variable>
             <role-name> ::= <role-name-constant> | <role-name-variable>
     <temporal-interval> ::= <temporal-interval-constant> | <temporal-interval-variable>
        <auth-statement> ::= admin says that <subject> can use the <role-atom>
                             during <temporal-interval>
       <query-statement> ::= admin asks does <subject> have <privilege> rights to <xpath-statement>
                             during <temporal-interval>
        <role-statement> ::= admin creates <role-atom>
             <role-atom> ::= role( <role-name>, <sign>, <xpath-statement>, <privilege> )
                  <sign> ::= + | -
       <xpath-statement> ::= in <document-name>, return <xpath-expressions>
         <document-name> ::= <document-name-constant> | <document-name-variable>
     <xpath-expressions> ::= <xpath-node> | <xpath-node>, <xpath-expressions>
            <xpath-node> ::= [ / ] <node-name> [<xpath-predicate>] /
             <node-name> ::= <node-name-constant> | <node-name-variable> | * | //
       <xpath-predicate> ::= <child-node-name> <predicate-relationship> <variable-value> |
                             <attribute-name> <predicate-relationship> <variable-value>
       <child-node-name> ::= <child-node-name-constant> | <child-node-name-variable>
        <attribute-name> ::= <attribute-name-constant> | <attribute-name-variable>
<predicate-relationship> ::= < | > | =
             <privilege> ::= read | write
```

Table 1. BNF for $\mathcal{A}^{xml(T)}$

Finally, section 5 briefly introduces an experimental software implementation of $\mathcal{A}^{xml(T)}$ before concluding the paper with some remarks.

2 Formal Language $\boldsymbol{\mathcal{A}^{xml(T)}}$

Our language, $\mathcal{A}^{xml(T)}$, consists of a finite set of predicate statements. These statements are used to create various rules in a security policy base. We present the syntax of our language in Backus-Naur Form (Table 1) with a definition of each element.

A *rule* is a conditional statement that allows the policy writer to specify a predicate statement to be validated based on the truth of other predicates. Rules include nonmonotonic reasoning derived through the absence of predicates. Our language also includes *deny-rule* statements which are for specifying conditional states that should never be allowed.

The *head-statement* from a *rule* consists of the predicate statements that will be validated true if the rules conditions are true as well. The head-statement itself can either be one of five statements; a *relationship-statement*, *grant-statement*, *query-statement*, *auth-statement*, or *role-statement*.

The *body-statement(s)* of a *rule* are the conditions that are reasoned upon to validate the *head-statement*. These are also made up of the same five statements used in the *head-statement*.

A *relationship-statement* confirms that some relationship between two objects in the security policy base are true. These relationships are represented by those predicate symbols found under the *relationship-atom*. There are a few *relationship-atoms* available that can be used in *relationship-statements*. Relationships for example could be hierarchical (below), mutually exclusive (separate), or be based on Allen's Temporal Interval relationships (during, starts, meets, etc.) [Allen, 1984].

The *role-statement* creates an access control role. The *role-atom* used in the statement includes a *role-name*, a *sign* which represents either positive or negative access to the object in question, an *xpath-statement* to identify an XML object, and finally the *privilege* that can be performed on the object.

An *xpath-statement* in $\mathcal{A}^{xml(T)}$ is a formal representation of an XPath expression. These expressions include the primary features of the syntax of XPath, such as single node queries, tree-like structured queries, wildcard queries, and predicate filters on nodes and attributes [WWW Consortium, 1999].

Grant-statements serve the purpose of assigning an access control *role* to a *subject* (a person requiring authorisation). This statement also includes a temporal argument to specify when the roles authorisation should be applied.

A *query-statement* is used when a query for subject authorisation is made. It represents the policy writers attempt to discover if a particular *subject* can perform the authorisation rights of a *role* at a specific *temporal-interval*.

Auth-statements specify that a *subject* who has been previously granted a *role* now has authorisation to access an object. We create rules in the policy base that will validate these statements by checking if a subject has positive authorisation to a role and that there are no conflicting rules. If these are true, then an *auth-statement* is created.

A *Policy Base*, which defines all the rights which subjects have over XML documents, is made up of *rule* statements and other *facts* about the access controlled environment. *Rules*, which can be written with variable arguments, are reasoned upon to determine when a subject is allowed to access a particular XML document (specified with an XPath expression). *Facts* are additional information such as <*relationship-statements*> that define role or temporal interval relations or <*grant-statements*> which specify that a subject be granted membership to a role. *Facts* aid in the reasoning of *rules*. We define a policy base as follows:

Definition 1. *A policy base D_A consists of finite facts and rules defining the access control rights that subjects have over XML objects in a database. Sub-*

jects, roles, XML objects, temporal intervals and all the relationships that exist between them exist within a domain and can be represented in D_A using the formal language $\mathcal{A}^{xml(T)}$.

2.1 Expression Examples with $\mathcal{A}^{xml(T)}$

In this section, we demonstrate utilising our formal language to express some common relationships and rules for a security policy base. In all cases, the $\mathcal{A}^{xml(T)}$ expressions have a similar natural language meaning.

Creating a temporal interval relationship The policy base writer can specify that the interval *morning_tea* is before *afternoon_tea* and that the interval *play_time* meets *nap_time*:

```
admin says before(morning_tea, afternoon_tea).
admin says meets(play_time, nap_time).
```

XML elements and attributes Using the *xpath-statement*, an arbitrary element named *cleaning_log* with the child element *cleaning_area* from the document "database.xml" can be represented like this:

```
in database.xml, return cleaning_log/cleaning_area
```

The policy writer can also specify more in the XPath by using predicates or wildcards. This *xpath-statement* uses a wildcard (*) to specify a single step between the elements *cleaning_information* and *cleaning_log*. The policy writer also uses a predicate expression to filter *cleaning_area*'s that have the attribute *type* equal to *office*.

```
in database.xml, return /janitor_logs/
cleaning_information/*/cleaning_log/cleaning_area[@type="office"]
```

Role Creation, Role Relationships, and Granting Authorisations The policy writer can create the *janitor* role. This role is allowed to *read* the element specified in our XPath from the previous example.

```
admin creates role(janitor, +, in database.xml, return /janitor_logs/
cleaning_information/*/cleaning_log/cleaning_area[@type="office"], read).
```

The policy writer can specify relationship statements between roles. They can state that the role *staff* is below the role *manager*, or in other words, is a child role, and that they also be mutually exclusive by specifying that they be separate.

```
admin says below(staff, manager).
admin says separate(staff, manager).
```

The policy writer can add a subject to a role's membership. For example, they can add the subject *tyler* to the role *janitor*. He will be able to access this role only during the *afternoon* temporal interval.

```
admin grants janitor to tyler during afternoon.
```

Here, the policy writer creates a complex rule stating that if any subject is a member of the role *janitor* during any time, then they should also be a member of the role *window_washers* during the same interval. The interval must also finish at the same time as *maintenance_time*. They add the condition that the subject also **not** be a member of the *electrician* role.

```
admin grants window_washer to SubX during TimeY
    if admin grants janitor to SubX during TimeY,
    admin says finishes(TimeY, maintenance_time),
    with absence admin grants electrician to SubX during TimeZ.
```

The *deny-rule* is useful for specifying rules where the validity of the *body-statements* are not desired. A deny-rule can be written to indicate that *patrick* should never be a member of the role *janitor* during any interval.

```
admin will deny if admin grants janitor to patrick during TimeY.
```

Query Statements The policy writer can query if a subject has the privilege to access an XML node(s) at a specific time. For example, they can check if *joel* can *read /a/b/c* during *morning*.

```
admin asks does joel have read rights to in example.xml, return /a/b/c during morning.
```

We have demonstrated some of the general expressiveness of $\mathcal{A}^{xml(T)}$. However, we have purposely only shown how to specify XML objects, roles, and subject membership to those roles. We have not explained how we know if or when a subject is allowed to perform the privileges given for a role. To do this, we must reason upon the policy base with a variety of rules. We discuss these rules in the next section.

2.2 Producing Authorisations with the XML Policy Base

With a security policy base D_A written in $\mathcal{A}^{xml(T)}$, it is possible to find which subjects have authorisations to what objects based on the roles they have been granted membership to. To do this, we reason upon statements that have been written in the policy base. The subject authorisations are found with a rule we refer to as the *authorisation rule*.

```
admin says that SUBJECT can use role(ROLE-NAME, +, XPATH, PRIVILEGE) during INTERVAL
    if admin grants ROLE-NAME to SUBJECT during INTERVAL,
    admin creates role(ROLE-NAME, +, XPATH, PRIVILEGE),
    with absence role(ROLE-NAME, -, XPATH, PRIVILEGE)
```

This rule is written to pertain to all *grant-statements*. It ensures that a role be positively authorised for use by a subject only if it does not conflict with a possible negative role with the same privileges and temporal interval (*conflict resolution*) [Ferraiolo et al., 1995]. If this rule produces an *auth-statement*, that is the indication that the subject in question does in fact have authorisation based on those specified in the *role-statement*.

Other defined rules like this applying to many aspects of our formal language must also be reasoned upon before authorisation is given to a subject. We refer to these as *language rules* within $\mathcal{A}^{xml(T)}$. They are discussed in more depth in the formal semantics of our language and are defined in two groups:

– *Role-based Access Control Rules* are included to ensure that features of the model are present (ie. separation of duty, conflict resolution, role propagation) and that authorisations are generated when querying the policy base (ie. the *authorisation rule*).
– *Temporal Interval Relationship Reasoning Rules* allow for defined temporal intervals to adhere to the relationships defined in Allen's work [Allen, 1984].

By using $\mathcal{A}^{xml(T)}$ to define a security policy base, we now have a determinable way to reason who has authorisation to what XML objects based on facts about subject privileges. However, to produce these authorisations and to also prove that our policy base written in $\mathcal{A}^{xml(T)}$ is satisfiable, we need a method to compute a result. To do this, we provide the semantics of our language in the form of an answer set program.

3 Semantics

We chose Answer Set Programming as the basis for our semantics because it provides the reasoning capabilities to compute the authorisations defined using our formal language. If properly translated, we can use an ASP solver (such as *smodels*) to find which authorisations will be validated true [Niemelä et al., 2000]. What we want to produce is an answer set that will have the same results as those produced from our formal language $\mathcal{A}^{xml(T)}$. We first present the alphabet of our ASP based language $\mathcal{A}_{LP}$ and then its formal semantics.

3.1 The Language Alphabet $\mathcal{A}_{LP}$

Entities Subjects, temporal intervals, role names, role properties, XPath's, and XPath properties make up the types of entities allowed in the language. These can either be constant or variable entities, distinguished by a lowercase or uppercase first letter respectively.

Function symbols

– `role(role-name, sign, isXPath(), priv)`, where *role-name* is the name of this role, *sign* is a + or − depending on if the role is allowing or disallowing a privilege, *isXPath* is an xpath predicate representing an element(s) from an XML document, and *priv* is the privilege that can be performed on the object (ie. read or write).
– `node(name, id, level, doc)`, represents a node in an XML document, where `name` is the name of that node (element), `id` is a distinct key in the document, `level` represents its hierarchical placement, and `doc` the document it originates from. We label each node with an ID and level for the purpose of distinguishing individual nodes. The reasoning behind this is based on various methods to do with *query rewriting*. This concept is presently beyond the scope of our work, however, we do direct you to their purposes in [Almendros-Jiménez et al., 2008,di Vimercati et al., 2005,Fan et al., 2004].
– `xpred(axis, query)`, represents an XPath predicate, where *axis* is the location of the node to apply the predicate *query* on.

Predicate symbols The first set of symbols are used for representing relationships between roles and temporal intervals. Their definitions are taken directly from $\mathcal{A}^{xml(T)}$.

below(role-name$_2$, role-name$_1$)	before(tempint$_2$, tempint$_1$)
separate(role-name$_2$, role-name$_1$)	overlap(tempint$_2$, tempint$_1$)
during(tempint$_2$, tempint$_1$)	meets(tempint$_2$, tempint$_1$)
starts(tempint$_2$, tempint$_1$)	equal(tempint$_2$, tempint$_1$)
finishes(tempint$_2$, tempint$_1$)	

This next set of symbols is used for defining and querying authorisations in the policy base and are also similar to their $\mathcal{A}^{xml(T)}$ equivalents.

```
grant(subject, role-name, tempint)
query(subject, isXPath(), priv, tempint)
auth(subject, isXPath(), priv, tempint)
```

A new predicate symbol is introduced in $\mathcal{A}_{LP}$ for conflict resolution reasoning on subject authorisations.

– `exist_neg(subject, xpath(), priv, tempint)` states that at least one negative `grant` for a *subject* exists.

And finally, four predicates are also introduced for providing relationships between XML nodes.

– `isXPath(node(), xpred())`, represents an XPath, consisting of a `node()` and `xpred()`.
– `isNode(node())`, indicates that the *node()* function exists.
– isParent(node$_2$(), node$_1$()), means *node*$_2$ is the parent or is hierarchically above *node*$_1$, where both are node functions.
– isLinked(node$_2$(), node$_1$()), means *node*$_2$ can be reached directly (is descended) from *node*$_1$, where both are node functions.
– `isAttr(attr_name, node())`, means *attr_name* is an XML attribute of the *node* function

In most cases, with an understanding of $\mathcal{A}^{xml(T)}$, the transformations and meanings of symbols and rules from $\mathcal{A}_{LP}$ are self explanatory (see Table 2).

$\mathcal{A}^{xml(T)}$	$\mathcal{A}_{LP}$
admin says below(doctor, admin_doctor).	below(doctor, admin_doctor).
admin says meets(open_shop, close_shop).	meets(open_shop, close_shop).
admin grants RoleX to SubY during TimeZ.	grant(SubY, RoleX, TimeZ).

Table 2. Transformation Examples

3.2 Formal Definitions

We define the semantics of our formal language by translating $\mathcal{A}^{xml(T)}$ into an answer set program. We refer to this translation as $Trans$. A *policy base*, D_A, is a finite set of rules and/or deny-rules, ψ, written in $\mathcal{A}^{xml(T)}$ as specified in Table 1. The generic rules, or *language rules*, for the same policy base, D_A, are a finite set of statements, α, written in $\mathcal{A}^{xml(T)}$.

α contains statements to provide:

- conflict resolution,
- separation of duty,
- role propagation,
- temporal interval relationship reasoning, and
- authorisation reasoning

Definition 2. *Let D_A be a policy base. We define $Trans(D_A)$ to be a logic program translated from D_A as follows:*

1. *for each rule or deny-rule, ψ, in D_A, $Trans(\psi)$ is in $Trans(D_A)$*
2. *for each statement α in D_A, $Trans(\alpha)$ is in $Trans(D_A)$*

A translated *rule* or *deny-rule*, $Trans(\psi)$, has the same form as those defined in Gelfond et al's Stable Model Semantics and answer set programming [Baral, 2003,Gelfond and Lifschitz, 1988]. A translated *rule* has the following form:

```
Trans(head-statement)_k ←
    Trans(body-statement)_k+1,...,
    Trans(body-statement)_m,
    not Trans(body-statements)_m+1,...,
    not Trans(body-statements)_n.
```

A translated *deny-rule* has the same form except for the dismissal of the *head-statement*.

The conflict resolution rules in α are located in the *authorisation rule* (Section 2.2). In $Trans(\alpha)$, conflict resolution rules are transformed into a new rule that checks if a subject has at least one negative grant for a role. We use this to reason if a conflict with a positive grant is possible. In $\mathcal{A}_{LP}$, exist_neg was introduced for this purpose. The translated rule is as follows:

```
exist_neg(S, isXPath(node(N, I, L, D), xpred(A, Q)), P, T) ←
    grant(S, R, T),
    role(R, -, isXPath(node(N, I, L, D),
    xpred(A, Q)), P).
```

Separation of duty in α is translated with a simple deny rule:

```
← grant(S, R1, T1), grant(S, R2, T2), separate(R2, R1).
```

Role propagation in α is also translated similarly in $Trans(\alpha)$ with two generic rules. The original rules were (1.) to do with transitivity between roles and (2.) for propagation of role properties. Their translation is as follows:

1. `below(R1, R3) ← below(R1, R2), below(R2, R3).`

2.
```
role(R1, Si, isXPath(node(N, I, L, D), xpred(A, Q)), P) ←
   below(R1, R2), role(R2, Si,
   isXPath(node(N, I, L, D), xpred(A, Q)), P)
```

α contains numerous rules that pertain to temporal interval relationship reasoning. Again, many of these rules are transformed from $\mathcal{A}^{xml(T)}$ to $\mathcal{A}_{LP}$ trivially. We show some of these rules from $Trans(\alpha)$:

Temporal Interval Containment Rule:
```
grant(S, R, T2) ←
   grant(S, R, T1), during(T2, T1).
```

Implicit Temporal Interval Relationships:
```
during(T2, T1) ← starts(T2, T1).
during(T2, T1) ← finishes(T2, T1).
before(T2, T1) ← meets(T2, T1).
```

Temporal Interval Transitive Relationships:
```
before(T1.T3) ← before(T1,T2), before(T2,T3).
during(T1.T3) ← during(T1,T2), during(T2,T3).
starts(T1.T3) ← starts(T1,T2), starts(T2,T3).
finishes(T1.T3) ← finishes(T1,T2), finishes(T2,T3).
equal(T1.T3) ← equal(T1,T2), equal(T2,T3).
```

Temporal Interval Bounded Rule:
```
during(T4, T1) ←
   starts(T2, T1), finishes(T3, T1),
   before(T2, T4), before(T4, T3).
```

Classical Negated Temporal Interval Rules:

As part of our semantics, we have also included classical negation in some aspects of the language. We use it in temporal interval reasoning for finding inconsistencies in the relationships defined in the policy base. For example, it is understood that if the predicate during(A, B) is true, then before(A, B) can not exist. The following rules are included to find this and any similar violations. We use the ¬ symbol to indicate classical negation of predicates.

```
¬before(T2.T1) ← during(T2,T1).         ¬meets(T2.T1) ← overlap(T2,T1).
¬overlap(T2.T1) ← during(T2,T1).        ¬equal(T2.T1) ← overlap(T2,T1).
¬meets(T2.T1) ← during(T2,T1).          ¬during(T2.T1) ← meets(T2,T1).
¬equal(T2.T1) ← during(T2,T1).          ¬overlap(T2.T1) ← meets(T2,T1).
¬during(T2.T1) ← before(T2,T1).         ¬equal(T2.T1) ← meets(T2,T1).
¬overlap(T2.T1) ← before(T2,T1).        ¬during(T2.T1) ← equal(T2,T1).
¬equal(T2.T1) ← before(T2,T1).          ¬before(T2.T1) ← equal(T2,T1).
¬during(T2.T1) ← overlap(T2,T1).        ¬overlap(T2.T1) ← equal(T2,T1).
¬before(T2.T1) ← overlap(T2,T1).        ¬meets(T2.T1) ← equal(T2,T1).
```

We did not define rules for `starts` and `finishes` as they will have been already implicitly included under the predicate `during`. We can use a *deny-rule* to ensure that these rules are enforced. The general rule would be like this:
```
← ¬during(T2.T1), during(T2,T1).
```

A *deny-rule* similar to this would be written for each temporal relationship predicate.

The Temporal Interval Equality Rule:

This rule simply ensures that any relationships for an interval that is `equal` to another will be repeated. For example, the rule would be written like the following for the predicate symbol `during`, and just like the classical negated rules, one would be written for every predicate in $\mathcal{A}_{LP}$.

during(T_3, T_2) $\leftarrow$ equal(T_1, T_2), during(T_3, T_1).

Finally, the *authorisation rule* (Section 2.3) in $\mathcal{A}^{xml(T)}$ is translated in $Trans(\alpha)$ as follows:

```
auth(S, isXPath(node(N, I, L, D), xpred(A, Q)), P, T) ←
    grant(S, R, T),
    role(R, +, isXPath(node(N, I, L, D),
    xpred(A, Q)), P),
    not exist_neg(S, isXPath(node(N, I, L, D),
    xpred(A, Q)), P, T).
```

A query on D_A, ϕ, written in $\mathcal{A}^{xml(T)}$ is a query statement, as specified in Table 1, and its translation, $Trans(\phi)$, is a `query` predicate from $\mathcal{A}_{LP}$.

Definition 3. *Let ϕ be a query on a policy base D_A written in $\mathcal{A}^{xml(T)}$. We define $Trans(\phi)$ as the translation of the query statement from $\mathcal{A}^{xml(T)}$ to $\mathcal{A}_{LP}$.*

An answer from a query ϕ is denoted as π and has the form of an authorisation statement, specified in Table 1, while its translation, $Trans(\pi)$, is an `auth` predicate from $\mathcal{A}_{LP}$.

Definition 4. *Let π be the answer from a query ϕ on policy base D_A written in $\mathcal{A}^{xml(T)}$. We define $Trans(\pi)$ as the translation of the authorisation statement from $\mathcal{A}^{xml(T)}$ to $\mathcal{A}_{LP}$.*

We define the relationship between our formal language and its translation into the semantics of answer set programming.

Definition 5. *Let D_A be a policy base, ϕ a query on it, and π the answer from that query. We say D_A entails π, or $D_A \models \pi$, iff for every answer set, S, of the logic program $Trans(D_A)$ with the query $Trans(\phi)$, $Trans(\pi)$ is in S.*

$D_A \models \pi$ iff $Trans(D_A) \models Trans(\pi)$

4 A Case Study

We will demonstrate the creation of a security policy for a scenario requiring access control to XML documents.

Scenario Description A hospital requires the implementation of an access control model to protect sensitive information it stores in a number of XML documents (see Figure 1). We will discuss roles created for four particular subjects at the hospital.

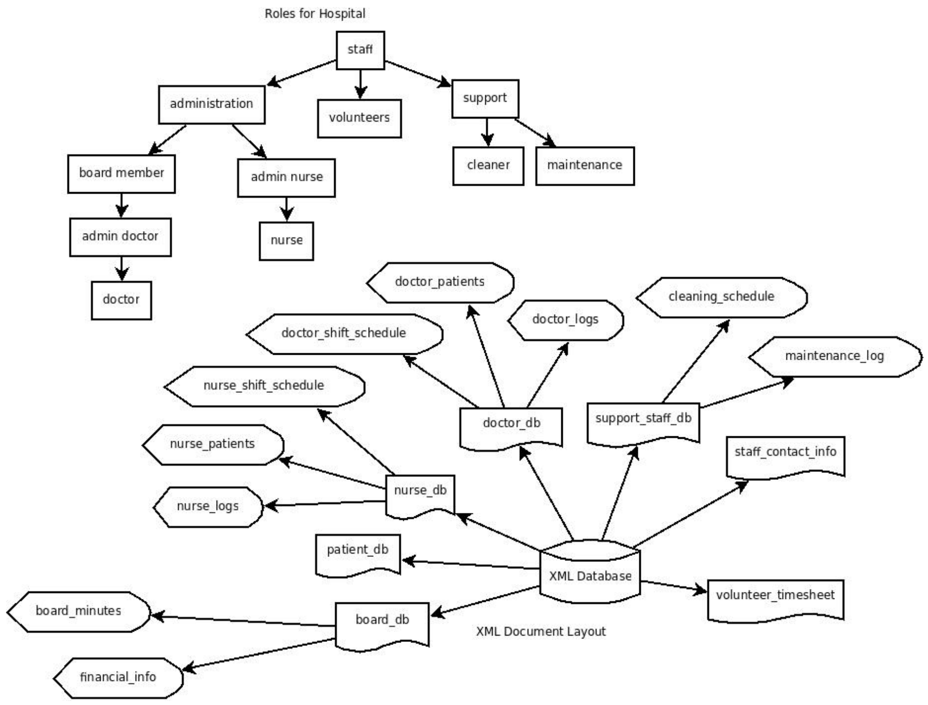

Fig. 1. Hospital Roles/XML Database Layout

Hospital Roles An *administration* role in the hospital will have access to read two nodes named *board_minutes* and *financial_info* from a document named *board_db*. Roles that are below the *administration* role will also inherit this rule. For example, a role named *board_member* will inherit these privileges. However, we will also include within the *board_member* role the privilege to write to the document.

The role *admin_doctor* will inherit its initial authorisations from *board_member*. *Admin_doctor* will be able to write to the *board_minutes* section of the *board_db* document. however, we will not allow them to write to the *financial_info* document.

In our policy base, we will allow the *admin_doctor* role to read and write to a few other documents. They will have access to read a *staff_contact_info* document and both read and write to the *patient_db* and *doctor_db* documents.

Finally, to demonstrate *separation of duty*, we will make the *admin_doctor* role mutually exclusive with the *administration* and *board_member* roles.

Table 3 shows these roles written in $\mathcal{A}^{xml(T)}$.

Policy Base Subjects and Rules Our first subject, Paul, will be granted membership to the *administration* role. Next, John will be a *board_member* and receive

```
Administration
 admin creates role(administration, +, in board_db, return /, read).
 admin says below(board_member, administration).
 admin creates role(board_member, +, in board_db, return /, write).
Administrative doctors
 admin says below(admin_doctor, board_member).
 admin says separate(admin_doctor, board_member).
 admin says separate(admin_doctor, administration).
 admin creates role(admin_doctor, -, in board_db, return /financial_info, write).
 admin creates role(admin_doctor, +, in staff_contact_info, return /, read).
 admin creates role(admin_doctor, +, in patient_db, return /, read).
 admin creates role(admin_doctor, +, in patient_db, return /, write).
 admin creates role(admin_doctor, +, in doctor_db, return /, read).
 admin creates role(admin_doctor, +, in doctor_db, return /, write).
```

Table 3. Hospital Roles

the same privileges as Paul in addition with being able to write to the *board_db* XML document. Both subjects will be granted their roles during the interval *wednesday.*

Lucy and Rita will both be members of the *admin_doctor* role. Lucy will utilize the privileges of the *admin_doctor* role for a single specific interval while Rita must be active in that same role at an interval directly following Lucy's. We will grant Lucy the *admin_doctor* role during *monday.*

The XML access control rules in which these subjects must abide to in the hospital are as follows. First, we will state relationships for some intervals like so:

```
admin says meets(monday, tuesday).
admin says meets(tuesday, wednesday).
admin says starts(midWeekMeeting, wednesday).
```

With these statements, the intervals *monday, tuesday, wednesday* and *midWeekMeeting* will be created. We can then make rules to specify during what temporal intervals our subjects should be granted membership to their roles.

```
admin grants administration to paul during wednesday.
admin grants board_member to john during wednesday.
admin grants admin_doctor to lucy during monday.
admin grants admin_doctor to rita during INT_J
    if admin grants admin_doctor to lucy during INT_I,
    admin says meets(INT_I, INT_J).
```

The last rule states that Rita be granted the role *admin_doctor* during a variable interval that must proceed Lucy's membership of the same role.

4.1 Logic Program Translation

With a completed policy base, we can translate all of the $\mathcal{A}^{xml(T)}$ rules into an $\mathcal{A}_{LP}$ answer set program. For our case study, we will demonstrate the translation of some of the policies for our subjects.

Role Translations From the defined roles, we will show the translation of some of the $\mathcal{A}^{xml(T)}$ rules.

This is the translation of the *administration* roles only privilege; the ability to read the *board_db* XML document.

```
role(administration, +, isXPath(node(/, 00, 0, board_db),
xpred(self, ‘‘’’)), read).
```

The XPath in this rule represents the *root* node of the *board_db* document, which is at the top-level (0) of the document and has the ID *00*.

This next role was originally written in $\mathcal{A}^{xml(T)}$ to specify that the *admin_doctor* not be allowed to write to the *financial_info* node in the *board_db* document. In $\mathcal{A}_{LP}$ it is written as:

```
role(admin_doctor, -, isXPath(node(/financial_info, ID, 1, board_db),
xpred(self, ‘‘’’)), write).
```

The XPath in this rule represents the *financial_info* node, with any *ID* (variable), at level (1) of the *board_db* document.

The following translated rules are for the hierarchical relationships of the roles and for the separation of duty for the *admin_doctor* role:

```
below(board_member, administration).
below(admin_doctor, board_member).
separate(admin_doctor, board_member).
separate(admin_doctor, administration).
```

The above `below` statements will in turn generate the following *language rules* for role propagation where it will be reasoned that *admin_doctor* is `below` *administration* due to transitive hierarchy and that the authorisation rights for those propagated roles should also be inherited.

```
below(admin_doctor, administration) ←
       below(admin_doctor, board_member), below(board_member, administration).
role(board_member, +, isXPath(node(/, 00, 0, board_db), xpred(self, ‘‘’’)), read) ←
    below(board_member, administration), role(administration, +,
    isXPath(node(/, 0a, 0, board_db), xpred(self, ‘‘’’)), read)
role(admin_doctor, +, isXPath(node(/, 00, 0, board_db), xpred(self, ‘‘’’)), write) ←
    below(admin_doctor, board_member), role(board_member, +,
    isXPath(node(/, 0a, 0, board_db), xpred(self, ‘‘’’)), write)
role(admin_doctor, +, isXPath(node(/, 00, 0, board_db), xpred(self, ‘‘’’)), read) ←
    below(admin_doctor, administration), role(administration, +,
    isXPath(node(/, 0a, 0, board_db), xpred(self, ‘‘’’)), read)
```

The `separate` predicates would generate *deny-rule* statements, however, due to space, we will just show the general rules with variables. In the real policy base, the variables representing the subjects and temporal intervals would be replaced with all those that exist. In our case, we are only concerned with the rules that pertain to subjects who may be members of *admin_doctor* and are attempting to join either *board_member* or *administration* or conversely.

```
← grant(S, admin_doctor, T1), grant(S, board_member, T2),
separate(admin_doctor, board_member).
...
← grant(S, admin_doctor, T1), grant(S, administration, T2),
separate(admin_doctor, administration).
...
```

Grant and Temporal Interval Relationship Translations We will now translate the rules granting subjects membership to roles and the temporal intervals we are using in the example. In $\mathcal{A}_{LP}$, they are straightforwardly translated as:

```
meets(monday, tuesday).
meets(tuesday, wednesday).
starts(midWeekMeeting, wednesday).
grant(paul, administration, wednesday).
grant(john, board_member, wednesday).
grant(lucy, admin_doctor, monday).
grant(rita, admin_doctor, INT_J)←
    grant(lucy, admin_doctor, INT_I),
    meets(INT_I, INT_J).
```

Implied Rules Because of the *language rules* of $\mathcal{A}^{xml(T)}$ and $\mathcal{A}_{LP}$, rules created in the policy base that agree with them may produce other implied rules as well. In this section, we will explain some of those implied rules.

```
during(midWeekMeeting, wednesday) ← starts(midWeekMeeting, wednesday).
grant(paul, administration, midWeekMeeting) ←
    grant(paul, administration, wednesday), during(midWeekMeeting, wednesday).
grant(john, board_member, midWeekMeeting) ←
    grant(john, board_member, wednesday), during(midWeekMeeting, wednesday).
```

The first rule is generated because of the *implicit temporal interval relationships* we incorporate into the formal language (Section 3.2). Briefly, if an interval *starts* or *finishes* another interval, then it is contained within it and a *during* relationship is implied. This relationship is seen with our intervals *midWeekMeeting* and *wednesday.*

The second and third rules, which demonstrate the *temporal interval containment rule*, use the new knowledge of *midWeekMeeting's* containment within *wednesday* to produce implied `grant` statements for Paul and John.

4.2 Experimenting with the $\mathcal{A}_{LP}$ program

We now present some examples of experimenting with the authorisations and rules of our $\mathcal{A}_{LP}$ policy base. The following queries and actions will be attempted:

1. Can Lucy write to the *financial information* XML node of the database on *monday*?
2. Can Rita read the *doctor database* XML node of the database on *tuesday*?
3. Can John read a node from the *board database* during the interval *midWeekMeeting.*
4. Can Paul read a node from the *patient database* node on *wednesday* if we grant him membership to the *admin_doctor* role?

We will explain the outcomes along with the $\mathcal{A}^{xml(T)}$ and $\mathcal{A}_{LP}$ statements made for each.

In query 1., we use the following $\mathcal{A}^{xml(T)}$ query to determine if Lucy can perform the action she is requesting.

admin asks does lucy have write rights to in board_db, return /financial_info during monday.

It is translated into its logic program equivalent like so:

```
query(lucy, isXPath(node(/financial_info, 01, 1, board_db), xpred(self, ‘‘’’)),
write, monday).
```

When the $\mathcal{A}_{LP}$ policy base is reasoned upon, the authorisation rule would determine that Lucy can in fact not do this action. This is because the *admin_doctor* role, which she is a member of, has a specific rule denying her this right. When the policy base is reasoned, an `exist_neg` predicate would be validated true for her. This would then in turn invalidate the authorisation rule hence denying her the `auth` predicate required to access the XML node. The following is the $\mathcal{A}_{LP}$ fragment highlighting the `exist_neg` predicate:

```
exist_neg(lucy, isXPath(node(/financial_info, 01, 1, board_db), xpred(self, ‘‘’’),
write, monday) ←
    grant(lucy, admin_doctor, monday),
    role(admin_doctor, -, isXPath(node(/financial_info, 01, 1, board_db),
    xpred(self, ‘‘’’)), write).
```

To reiterate its meaning, because a `role` statement disallowing *admin_doctors* from *writing* to the *financial_info* node exists and a `grant` for this role exists for Lucy on *monday*, an `exist_neg` will be generated.

In query 2., the following $\mathcal{A}^{xml(T)}$ and $\mathcal{A}_{LP}$ queries are written for Rita:

admin asks does rita have read rights to in doctor_db, return / during tuesday.

It is translated into its logic program equivalent like so:

```
query(rita, isXPath(node(/, 00, 0, doctor_db), xpred(self, ‘‘’’)),
read, tuesday).
```

In this case, in our original policy base we did not write that Rita have membership to the *admin_doctor* on any specific temporal intervals (her membership would imply the privilege to *read* the database). We did however write a rule that specified she be granted the role during a temporal interval that directly followed one where Lucy was a member of the role. Upon reasoning the policy base, that rule would generate a `grant` statement for Rita like so:

```
grant(rita, admin_doctor, tuesday)←
    grant(lucy, admin_doctor, monday),
    meets(monday, tuesday).
```

Note the replacement of the variable temporal intervals with ones that would conclude with Rita being a member of the *admin_doctor* role. This rule would consequently produce an `auth` predicate for Rita since no other rules in the policy base would conflict with it. For the purpose of completion, here is the authorisation rule for Rita:

```
auth(rita, isXPath(node(/, 00, 0, doctor_db), xpred(self, ‘‘’’)), read, tuesday) ←
    grant(rita, admin_doctor, tuesday),
    role(admin_doctor, +, isXPath(node(/, 00, 0, doctor_db),
    xpred(self, ‘‘’’)), read),
    not exist_neg(rita, isXPath(node(/, 00, 0, doctor_db),
    xpred(self, ‘‘’’)), read, tuesday).
```

Our `query` matches an `auth` predicate found in the reasoned policy base. Therefore, we use this knowledge to determine that although it was not directly specified, Rita can in fact read the *doctor_db* on *tuesday*.

Query 3. has a similar result to query 2. Although it is not directly specified in the policy base, it will be determined that John can read the *board_db* XML document during the interval *midWeekMeeting*. The queries are as follows:

admin asks does john have read rights to in board_db, return / during midWeekMeeting.

It is translated into its logic program equivalent like so:

```
query(john, isXPath(node(/, 00, 0, board_db), xpred(self, ‘‘’’)),
read, midWeekMeeting).
```

We already showed in the previous section that because of the *language rules* of our formal language some *implied* rules are generated when the policy base is reasoned. We wrote that John should be a member of the *board_member* role during *wednesday*. This query is however concerned with the interval *midWeekMeeting*. Fortunately, that interval *starts wednesday*. In our language, we specified that the relationship *starts* implies the relationship *during* as well. This implication therefore produces a `during` predicate for the two intervals. Our policy base is further reasoned upon to determine that the following circumstances suffice for the *temporal interval containment rule* to produce a `grant` statement for John giving him membership to *board_member* during *midWeekMeeting* as well as *wednesday*.

Therefore, with the implied `grant` statement present and the absence of any conflicting rules, the authorisation rule would produce an `auth` predicate allowing John to read the node similar to the one for query 2.

Finally, in query 4, the principle of *separation of duty* is demonstrated. We actually can dispense with the query in this example because the action of attempting to grant Paul membership to the role *admin_doctor* will produce a fault in our policy base therefore making the query pointless.

Originally, we granted Paul membership to the role *administration*. However, we also stated that this role be *separate* from the *admin_doctor* role. We already discussed the generation of *deny-rules* for separation of duty in the previous section. In this case, if we attempted to include the statement

```
grant(paul, admin_doctor, wednesday).
```

the following *deny-rules* body would validate as true and therefore invalidate the whole policy base.

```
← grant(paul, admin_doctor, wednesday), grant(paul, administration, wednesday),
separate(admin_doctor, administration).
```

As soon as the violating `grant` statement is present, the policy base is considered incorrect and any queries followed ignored until the violation is corrected.

This case study has demonstrated a minor amount of the expressive power of $\mathcal{A}^{xml(T)}$ and $\mathcal{A}_{LP}$. We were able to show some simple examples of access control using the language rules and principles incorporated into it. However, we only just touched upon areas of the language such as the temporal interval reasoning rules and role propagation. In any case, a general understanding of the formal language should have been gained.

5 Consideration for an Implementation

Presently, we have an initial implementation to test the true expressiveness, capability, and limitations of $\mathcal{A}^{xml(T)}$ [Policarpio and Zhang, 2010].

This implementation is only an experimental prototype, so we intentionally limited its development so that it was strictly capable of only doing the following primary functions:

- $\mathcal{A}^{xml(T)}$ policy base management (adding, editing, deletion of rules)
- translation of the $\mathcal{A}^{xml(T)}$ policy base into an $\mathcal{A}^{LP}$ logic program

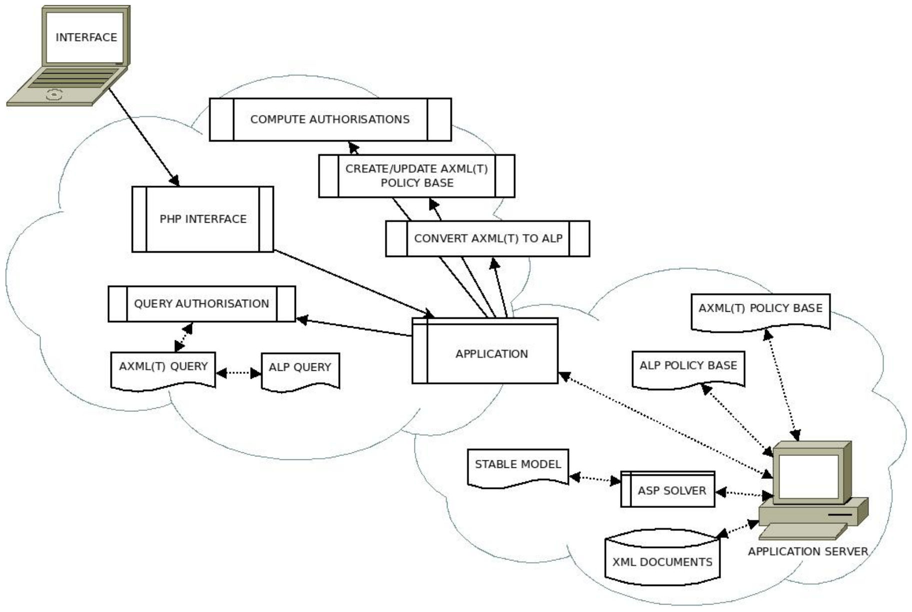

Fig. 2. System Structure

– computation of authorisations
– querying of the computed authorisations to discover what privileges a subject has during some interval

Eventually, this prototype will be able to be extended so that it provides a full fledged access controlled XML environment, however, at this point we were currently only focused on examining $\mathcal{A}^{xml(T)}$'s feasibility as an access control model and not as real world application. Our plan was to have the prototype perform and produce the same actions and results we presented in our case study.

The high-level structure of the prototype application is shown in Figure 2. We designed a management module called *pb_mgr* (policy base manager) that contains a majority of the functionality mentioned. For ease of use, we incorporated a web-based user interface to execute the module. Besides the functionality already mentioned, note the presence of an ASP solver and XML Documents database in Figure 2. The Answer Set Program solver represents the tools we use to ground the variables in the translated policy base and also compute a stable model from it. For the sake of simplicity, and because this is only an experimental implementation, we stored the XML documents in a local directory rather than a sophisticated XML storage system.

The policy base manager *pb_mgr* is written in the Python scripting language while the web interface was written in PHP. We setup a local Linux server with the Apache HTTP Server Project (httpd 2.2.15), PHP (5.3.2), and Python (2.6.5).

We tested the prototype with various policy base scenarios, similar to the one shown in our case study, to ensure that everything was working properly. Expectedly, $\mathcal{A}^{xml(T)}$ performed well in terms of the basic features and principles we incorporated into its formalisation and semantic translation. We did encounter various implementation difficulties, however, for an in-depth explanation of the prototype, its design, and the solutions to those difficulties, please refer to our implementation paper [Policarpio and Zhang, 2010].

6 Conclusion

In this paper, we presented a formal language of authorisation for XML documents. We demonstrated its expressive power to provide role-based access control with temporal constraints. We provided a semantic definition through the translation of the high level language into an answer set program. We presented a case study that defined some security policies in $\mathcal{A}^{xml(T)}$, translated them into an $\mathcal{A}_{LP}$ logic program, and then computed the output from some queries and actions performed on it. Finally, we briefly discussed our initial experiment with an $\mathcal{A}^{xml(T)}$ software implementation.

In our continued research, we are looking into the concept of *query containment*. It may have been noticed, but the examples used in this paper do not fully consider the fact that further authorisations may also be implied when the results of an XPath query are contained within another XPath used in an `auth` statement. We briefly touched upon this idea with the *temporal interval containment rule*. In either case, if it can be determined that containment is present, then authorisations should be understood or generated for the contained results. Query containment is useful because it allows us to bypass the reasoning required for contained queries. If we can determine (1) the containment of queries and (2) the validity of the container query then a considerable amount of work can be eliminated by deducing that (3) the contained queries are also valid.

We plan to present a newly updated formal language that will incorporate this idea as a feature.

References

[Allen, 1984] Allen, J. F. (1984). Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154.

[Almendros-Jiménez et al., 2008] Almendros-Jiménez, J. M., Becerra-Terón, A., and Enciso-ba Nos, F. J. (2008). Querying xml documents in logic programming*. *Theory Pract. Log. Program.*, 8(3):323–361.

[Anutariya et al., 2003] Anutariya, C., Chatvichienchai, S., Iwaihara, M., Wuwongse, V., and Kambayashi, Y. (2003). A rule-based xml access control model. In *RuleML*, pages 35–48.

[Apache Software Foundation, 2010] Apache Software Foundation (2010). The apache http server project. http://httpd.apache.org/.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

[Bertino et al., 1998] Bertino, E., Bettini, C., Ferrari, E., and Samarati, P. (1998). An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3):231–285.

[Bertino et al., 2000] Bertino, E., Braun, M., Castano, S., Ferrari, E., and Mesiti, M. (2000). Author-x: A java-based system for xml data protection. In *In IFIP Workshop on Database Security*, pages 15–26.

[Bertino et al., 2004] Bertino, E., Carminati, B., and Ferrari, E. (2004). Access control for xml documents and data. *Information Security Technical Report*, 9(3):19–34.

[Damiani et al., 2002] Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. (2002). A fine-grained access control system for xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(2):169–202.

[di Vimercati et al., 2005] di Vimercati, S. D. C., Marrara, S., and Samarati, P. (2005). An access control model for querying xml data. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 36–42, New York, NY, USA. ACM.

[Fan et al., 2004] Fan, W., Chan, C., and Garofalakis, M. (2004). Secure xml querying with security views. In *SIGMOD, 2004: Proceedings of the 2004 ACM SIGMOD international conference on Management Data*. ACM Press.

[Ferraiolo et al., 1995] Ferraiolo, D. F., Cugini, J. A., and Kuhn, D. R. (1995). Role-based access control (rbac): Features and motivations. In *11th Annual Computer Security Applications Proceedings*.

[Gabillon, 2005] Gabillon, A. (2005). A formal access control model for xml databases. In *Lecture notes in computer science, 3674*, pages 86–103.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K., editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts. The MIT Press.

[Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming? In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 1594–1597. AAAI Press.

[Niemelä et al., 2000] Niemelä, I., Simons, P., and Syrjänen, T. (2000). Smodels: a system for answer set programming. In *Proceedingsof the 8th International Workshop on Non-Monotonic Reasoning*.

[Policarpio and Zhang, 2009] Policarpio, S. and Zhang, Y. (2009). A formal language for specifying complex xml authorisations with temporal constraints. In *Inscrypt 2009: The 5th China International Conference on Information Security and Cryptology*, pages 169–183, Beijing, China. State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences, Beijing , China.

[Policarpio and Zhang, 2010] Policarpio, S. and Zhang, Y. (2010). An implementation of $\mathcal{A}^{xml(T)}$: A formal language of authorisation for xml documents. Manuscript.

[Python Software Foundation, 2010] Python Software Foundation (2010). Python programming language. http://www.python.org/.

[The PHP Group, 2010] The PHP Group (2010). Php: Hypertext preprocessor. http://www.php.net/.

[WWW Consortium, 1999] WWW Consortium (1999). Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath.

[WWW Consortium, 2008] WWW Consortium (2008). Extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/REC-xml/.