A Formal Language for Specifying Complex XML Authorisations with Temporal 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 (XML) is utilised in many Internet applications we are using today. However, as with many computing technologies, vulnerabilities exist in XML that can allow for malicious and unauthorised use. Applications that utilise XML are therefore susceptible to security faults if they do not provide their own methods. Our research focuses on developing a formal language which can provide access control to information stored in XML formatted documents. This formal language will have the capacity to reason if access to an XML document should be allowed. Our language, $\mathcal{A}^{xml(T)}$, allows for the specification of authorisations on XML documents based on the popular Role-based Access Control model. Temporal interval reasoning is the study of logically representing time intervals and relationships between them. As part of our research, we have also included this aspect in our language $\mathcal{A}^{xml(T)}$ because we believe it will allow us to specify even more powerful access control authorisations.

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

1 Introduction

Many applications utilise the Extensible Markup Language [9] as a tool to store and retrieve information. However, the guarantee that information stored in XML documents is secure and is only accessible by authorised users is not possible unless an external method is used. XML does not have any inherent security methods as part of its specification [9]. An XML document is essentially a formatted plain text file that can be freely viewed and edited. Therefore there is a demand for methods in which access to XML documents can be controlled.

In this paper, we present our work on the development of a formal language that will provide access control to XML documents. We incorporate the XML query language, XPath [8], into our formal language so that we can define which documents (or elements within a document) we would like to restrict access to.

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
```

Our formal language uses the Role-based Access Control model [15] 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. It also allows us to include features like separation of duty and conflict resolution directly into the language [15].

Finally, we include Allen's Temporal Interval Relationship logic [1]. 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. We include this aspect of temporal reasoning in our language so that we can specify time constraints on authorisations to designate when they should be applied.

We utilise the formal language to produce a *security policy base*. The policy base contains all the $\mathcal{A}^{xml(T)}$ rules of authorisation for the XML documents requiring access control. The policy base can be reasoned upon to determine which authorisations should be followed.

The rest of this 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 to a logic program under answer set semantics. In section 4, an example is also presented to show the application of $\mathcal{A}^{xml(T)}$ in XML authorisation specification and reasoning. Section 5 briefly discusses the related work. Finally, Section 6 concludes the paper with some remarks.

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

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. The statements are written from the point of view of the policy base writer or *admin*. This single subject represents the author of the access control policies.

2.1 Syntax

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 specifiying conditional states that should never be allowed.

```
                  <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> |
                             <request-statement> | <auth-statement> |
                             <role-statement>
       <body-statements> ::= <body-statement> | <body-statement>, <body-statements>
        <body-statement> ::= <relationship-statement> | <grant-statement> |
                             <request-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>
     <request-statement> ::= admin asks is <subject> a member of <role-name>
                             during <temporal-interval>
        <auth-statement> ::= admin says that <subject> can use the <role-atom>
                             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 $\mathcal{A}^{xml(T)}$

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*, *request-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 is 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.) [1].

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 [8].

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 *request-statement* is used when a query for subject authorisation is made. It represents the policy writers attempt to discover if a particular *subject* is a member 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.

2.2 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.

Creating a temporal interval relationship The policy base writer specifies 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 the 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 creates 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 specifies 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 adds a subject to a roles membership. They 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.
```

Request Statements The policy writer can query if a subject is a member of a role at a specific time. They will check if *taro* is a member of the *musician* role during *rehearsals*.

```
admin asks is taro a member of musicians during rehearsals.
```

2.3 Producing Authorisations and Querying the XML Policy Base

With a security policy base 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 asks is SUBJECT a member of ROLE-NAME 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 postively 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* [15]). 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*.

There are other defined rules like this that apply to many aspects of our formal language that 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 [1].

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 [4] 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* [18]) to find which authorisations will be validated true. 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, XPaths, 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, xpath(), 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, *xpath* is an xpath function 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.
- `xpred(axis, query)`, represents an XPath predicate, where *axis* is the location of the node to apply the predicate *query* on.
- `xpath(node(), xpred())`, represents an XPath, consisting of a `node()` and `xpred()`.

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-name₂, role-name₁)
separate(role-name₂, role-name₁)
during(tempint₂, tempint₁)
starts(tempint₂, tempint₁)
finishes(tempint₂, tempint₁)
before(tempint₂, tempint₁)
overlap(tempint₂, tempint₁)
meets(tempint₂, tempint₁)
equal(tempint₂, tempint₁)
```

These next set of symbols are 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)
request(subject, role-name, tempint)
auth(subject, xpath(), 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.

- `isNode(node())`, indicates that the *node()* function exists.
- `isParent(node₂(), node₁())`, means *node*$_2$ is the parent or is hierarchically above *node*$_1$, where both are node functions.
- `isLinked(node₂(), node₁())`, 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. However, extra consideration must be given to the method in which XPaths are handled in $\mathcal{A}_{LP}$.

3.2 Handling XPaths in $\mathcal{A}_{LP}$

There was a problem handling particular XPaths in $\mathcal{A}_{LP}$ because our formal language implements features and syntax of XPath that are difficult to translate and use in answer set programming. Specifically, this is the use of wildcards and predicate queries. We refer to these problematic XPaths as *dynamic* XPaths because they can represent zero to many XML nodes, as opposed to *static* XPaths which can only refer to zero or one node. When dynamic XPaths are used in a logic program, it is difficult to specify which nodes should have authorisation rules applied to them. This is because the nodes that are meant to be represented by the XPath are not yet known. With static XPaths, it is clearly stated what node requires authorisation.

We consider the approach by Almendros-Jimenez et al. [2] which described a method to represent the structure and data of an XML document as a logic program. Using a similar technique, we provide new rules that *rewrite* dynamic XPaths into static ones. The idea of *query rewriting* is to use the structure of XML documents to understand and define what dynamic XPaths might mean [13, 14].

These concepts are what make up the majority of the language rules for *XPath Translation* in the policy base. We introduced the functions `node`, `xpred`, and `xpath`, and the predicate symbols `isNode`, `isParent`, `isLinked` and `isAttr` into $\mathcal{A}_{LP}$ for this reason. These various new functions and predicates allow us to write rules to satisfy different kinds of dynamic XPaths. However, due to space constraints, we will show only one example of an XPath transformation rule.

XPath Tranformation Rule For this example, we will assume that the structure (schema) of the XML documents are already defined using the `node` function and predicate statements introduced earlier.

This rule will determine the meaning of a wildcard (*) in an XPath like this `/A/*/C`. It will produce an `xpath` that will return the element C which can only be reached from the parent element A.

```
xpath(node(C, ID3, 3, DOC), xpred(self, ‘‘’’)) ←
   isParent(
   node(X, ID2, 2, DOC),
   node(C, ID3, 3, DOC)),
   isParent(
   node(A, ID1, 1, DOC),
   node(X, ID2, 2, DOC)).
```

The rule determines if an arbitrary node, X, is the parent of node C and is the child of node A. In each `node`, we specify the `level` and use variables for the `id` and `document`. This allows for the rule to determine all possible XPaths that satisfy the relationship conditions. For those `nodes` that satisify the rule, we can produce an XPath function for C that we can guarantee is meant to be produced from the XPath `/A/*/C`.

Remarks It is important to note that these rules are not part of our semantics, which will be presented in the next section. This is because the process of transforming the XPaths occur at an intermediate level that is not concerned with the reasoning of the security policy base. Also, because these rules are written specific to varying XPaths, it is not possible to give a formal definition of their translation. We consider the transformations of XPaths as happening before the translation of an $\mathcal{A}^{xml(T)}$ policy base into $\mathcal{A}_{LP}$.

3.3 Formal Definitions

We define the semantics of our formal language by translating $\mathcal{A}^{xml(T)}$ into an answer set logic 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 1. *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's Stable Model Semantics [17] and answer set programming [4]. 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 2.3). 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, xpath(node(N, I, L, D), xpred(A, Q)), P, T) ←
    grant(S, R, T),
    role(R, -, xpath(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).
```

If `grant` predicates are giving a subject membership to roles that are stated as being `separate`, then the statement should be denied and the logic program faulted.

Role propagation in α is also translated similarily 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. The propagation rule searches for roles that are hierarchically related (using `below`) and copies the authorisation properties from the parent role to the descendent role. Their translation is as follows:

```
1. below(R1, R3) ← below(R1, R2), below(R2, R3).
2. role(R1, Si, xpath(node(N, I, L, D), xpred(A, Q)), P) ←
   below(R1, R2), role(R2, Si,
   xpath(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 $Trans(\alpha)$ [1]:

Temporal Interval Bounded Rule:

This rule searches for an interval that is contained within another but does not overlap the beginning or end of the outer interval.

```
during(T4, T1) ←
   starts(T2, T1), finishes(T3, T1),
   before(T2, T4), before(T4, T3).
```

Temporal Interval Containment Rule:

If the temporal interval used in a `grant` predicate is found to have an interval contained within it, then a similar `grant` will be applied to the subject for that contained interval.

```
grant(S, R, T2) ←
   grant(S, R, T1), during(T2, T1).
```

Implicit Temporal Interval Relationships:

These rules apply some implied temporal relationships we have choosen to implement for our own purposes. Namely, that `starts` and `finishes` should imply `during` and that `meets` should imply `before`.

```
during(T2, T1) ← starts(T2, T1).
during(T2, T1) ← finishes(T2, T1).
before(T2, T1) ← meets(T2, T1).
```

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

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

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

[1] All rules can be found in original manuscript

Definition 2. *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 request 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 3. *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 4. *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 An Example

We will demonstrate the creation of a security policy for a scenario requiring access control to XML documents. Due to space constraints, we will only have a very small example and also forgo XPath's utilising predicates.

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

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 be able to write to the *board_minutes* section of the *board_db* 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.

Table 2 shows these roles written in $\mathcal{A}^{xml(T)}$.

Policy Base Subjects and Rules Within our case scenario, we will focus on two subjects, Lucy and Rita, both administrative doctors. Lucy will utilize the

```
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, administration).
 admin creates role(admin_doctor, +, in board_db, return /board_minutes, 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 2. Hospital Roles

privileges of the *admin_doctor* role for a single specific interval while Rita must be active in that same role at any other time.

The XML access control rules in which these subjects must abide to in the hospital are as follows.

For Lucy, we can specify that she has the *admin_doctor* role on some arbitary interval *tuesday*. We could state some relationships for this interval like so:

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

Now we can make a rule that states that Lucy be active with the *admin_doctor* role on *tuesday* and Rita be active during any other time. In this case, possibly *monday and wednesday* or any other interval defined in the policy base.

```
admin grants admin_doctor to lucy during tuesday.
admin grants admin_doctor to rita during INT_I
     if with absence admin grants admin_doctor to lucy during INT_I.
```

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 our policies for Lucy and Rita.

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

This rule was orignally written in $\mathcal{A}^{xml(T)}$ to specify that the *admin_doctor* role be allowed to write to the *board_minutes* node in the *board* database. In $\mathcal{A}_{LP}$ it is written as:

```
role(admin_doctor, +, xpath(node(/board_minutes, ID, 0, board_db), xpred(self, ‘‘’’)),
write).
```

As mentioned earlier, we will forgo the explanation of the translation of XPaths. However, briefly, this XPath represents the /board_minutes node, with any *ID*, at the top-level (0) of the *board_db* document.

Grant Translations We will now translate some rules granting subjects membership to roles. We take a look at the rule that specified that Lucy be granted the *admin_doctor* on the interval *tuesday* and that Rita have it at any other interval. In $\mathcal{A}_{LP}$, they are translated as:

```
meets(monday, tuesday).
meets(tuesday, wednesday).
grant(lucy, admin_doctor, tuesday).
grant(rita, admin_doctor, I) ← not grant(lucy, admin_doctor, I).
```

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

We now present authorisation queries on our policies. We will show the queries and results in $\mathcal{A}^{xml(T)}$ and $\mathcal{A}_{LP}$.

We first create requests of authorisation for Lucy and Rita to obtain the role *admin_doctor* during the temporal interval *tuesday*.

In $\mathcal{A}^{xml(T)}$, we would write request-statements like so:

admin asks is lucy a member of admin_doctor during tuesday.
admin asks is rita a member of admin_doctor during tuesday.

In $\mathcal{A}_{LP}$, those statements are translated into the following `request` statements:

```
request(lucy, admin_doctor, tuesday).
request(rita, admin_doctor, tuesday).
```

If we were to logically reason upon our $\mathcal{A}^{xml(T)}$ policy base with the previous request-statements, the following statements would be produced by the *authorisation rule* (Section 2.3).

admin says that lucy can use role(admin_doctor, +, in board_db, return /, read) during tuesday.

admin says that lucy can use role(admin_doctor, +, in board_db, return /board_minutes, write) during tuesday.

admin says that lucy can use role(admin_doctor, +, in staff_contact_info, return /, read) during tuesday.

admin says that lucy can use role(admin_doctor, +, in patient_db, return /, read) during tuesday.
admin says that lucy can use role(admin_doctor, +, in patient_db, return /, write) during tuesday.
admin says that lucy can use role(admin_doctor, +, in doctor_db, return /, read) during tuesday.
admin says that lucy can use role(admin_doctor, +, in doctor_db, return /, write) during tuesday.

When our translated policy base and `request` predicates are computed in a stable model solver, an answer set containing the following equivalent facts is generated.

```
auth(lucy, xpath(node(/, idbdb00, 0, board_db), xpred(self, ‘‘’’), read, tuesday).

auth(lucy, xpath(node(/board_minutes, idbdb01, 0, board_db),
xpred(self, ‘‘’’), write, tuesday).

auth(lucy, xpath(node(/, idscidb00, 0, staff_contact_info),
xpred(self, ‘‘’’), read, tuesday).

auth(lucy, xpath(node(/, idpdb00, 0, patient_db), xpred(self, ‘‘’’), read, tuesday).
auth(lucy, xpath(node(/, idpdb00, 0, patient_db), xpred(self, ‘‘’’), write, tuesday).
auth(lucy, xpath(node(/, idddb00, 0, doctor_db), xpred(self, ‘‘’’), read, tuesday).
auth(lucy, xpath(node(/, idddb00, 0, doctor_db), xpred(self, ‘‘’’), write, tuesday).
```

Because Rita's rule specifies that she cannot be a member of *admin_doctor* at any interval that Lucy is, her requests for authorisation do not generate any results. Lucy however retrieves all the authorisations that an *admin_doctor* should. Referring to Table 2, we can identify all the *admin_doctor* roles that are used to create the auth-statements and `auth` predicates above.

5 Related Work

Damiani et al. [11, 12] provided some essential work in the field of XML access control. In [12], a fine-grained access control model is discussed. This model takes an XML document and designates access rights on each element. They provide implementations of rule propagation and also include features such as both positive and negative authorisations and conflict resolution 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 [12].

Crampton [10] utilised the role-based access control model for specific use with XML documents. He utilises the same object-based approach by using the XPath query language. However, Crampton points out that his work only focuses on reading XML documents [10].

In [5, 7], Bertino et al. 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 (however, 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 [5, 7]. 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 [6]. 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 [3, 16]. 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.

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. Finally, we gave a brief example of defining a security policy base in $\mathcal{A}^{xml(T)}$, translating it into an $\mathcal{A}_{LP}$ logic program, and then computing the authorisations from it.

Acknowledgement This research is supported in part by an Australian Research Council Linkage grant LP0883646.

References

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

2. J. M. Almendros-Jiménez, A. Becerra-Terón, and F. J. Enciso-ba Nos. Querying xml documents in logic programming*. *Theory Pract. Log. Program.*, 8(3):323–361, 2008.
3. Chutiporn Anutariya, Somchai Chatvichienchai, Mizuho Iwaihara, Vilas Wuwongse, and Yahiko Kambayashi. A rule-based xml access control model. In *RuleML*, pages 35–48, 2003.
4. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
5. E. Bertino, M. Braun, S. Castano, E. Ferrari, and M. Mesiti. Author-x: A java-based system for xml data protection. In *In IFIP Workshop on Database Security*, pages 15–26, 2000.
6. Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3):231–285, 1998.
7. Elisa Bertino, Barbara Carminati, and Elena Ferrari. Access control for xml documents and data. *Information Security Technical Report*, 9(3):19–34, July-September 2004.
8. The WWW Consortium. Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath, 1999.
9. The WWW Consortium. Extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/REC-xml/, November 2008.
10. Jason Crampton. Applying hierarchical and role-based access control to xml documents. In *SWS '04: Proceedings of the 2004 workshop on Secure web service*, pages 37–46, New York, NY, USA, 2004. ACM.
11. E. Damiani, S. De, Capitani Vimercati, S. Paraboschi, and P. Sarnarati. Securing xml documents. In *In Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000*, pages 121–135. Springer, 2000.
12. Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(2):169–202, 2002.
13. Sabrina De Capitani di Vimercati, Stefania Marrara, and Pierangela Samarati. 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, 2005. ACM.
14. W. Fan, C. Chan, and M. Garofalakis. Secure xml querying with security views. In *SIGMOD, 2004: Proceedings of the 2004 ACM SIGMOD international conference on Management Data*. ACM Press, 2004.
15. D. F. Ferraiolo, J. A. Cugini, and D. Richard Kuhn. Role-based access control (rbac): Features and motivations. In *11th Annual Computer Security Applications Proceedings*, 1995.
16. Alban Gabillon. A formal access control model for xml databases. In *Lecture notes in computer science, 3674*, pages 86–103, 2005.
17. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
18. I. Niemelä, P. Simons, and T. Syrjänen. Smodels: a system for answer set programming. In *Proceedingsof the 8th International Workshop on Non-Monotonic Reasoning*, April 2000.