Towards Attribute-Based Authorisation for Bidirectional Programming

Lionel Montrieux
lionel@nii.ac.jp

Zhenjiang Hu
hu@nii.ac.jp

National Institute of Informatics
Tokyo, Japan

ABSTRACT

Bidirectional programming allows developers to write programs that will produce transformations that extract data from a source document into a view. The same transformations can then be used to update the source in order to propagate the changes made to the view, provided that the transformations satisfy two essential properties.

Bidirectional transformations can provide a form of authorisation mechanism. From a source containing sensitive data, a view can be extracted that only contains the information to be shared with a subject. The subject can modify the view, and the source can be updated accordingly, without risk of release of the sensitive information to the subject. However, the authorisation model afforded by bidirectional transformations is limited. Implementing an attribute-based access control (ABAC) mechanism directly in bidirectional transformations would violate the essential properties of well-behaved transformations; it would contradict the principle of separation of concerns; and it would require users to write and maintain a different transformation for every subject they would like to share a view with.

In this paper, we explore a solution to enforce ABAC on bidirectional transformations, using a policy language from which filters are generated to enforce the policy rules.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls; H.2.7 [**Database Administration**]: Security, integrity, and protection

General Terms

Security

Keywords

authorization, access control, bidirectional transformation

SACMAT'15, June1–3, 2015, Vienna, Austria.

ACM 978-1-4503-3556-0/15/06 ...$15.00.
http://dx.doi.org/10.1145/2752952.2752963.

1. INTRODUCTION

A bidirectional transformation is a pair of functions, *get* and *put* that maintain consistency between a source and a view [10]. The source and view are typically graphs, or structured documents such as XML documents. The *get* function takes a source document, and produces a view document. On the contrary, the *put* (or *putback*) function takes both the source and an updated view, and produces an updated source, where the changes made to the view have been propagated. The transformation from source to view is called a *forward* transformation, while the transformation from view to source is called a *backward* transformation.

Bidirectional transformations have recently received a lot of attention, both from the programming languages community [2, 4, 11, 17, 20, 36, 27] and from the software engineering community, where they have been used in contexts as varied as model-driven engineering [19, 35], consistent website updating [25], parallel programming [24], and many others [6]. Since 2012, a workshop dedicated to bidirectional transformations is organised every year [16].

A particularly interesting class of bidirectional transformations is the class of *well-behaved* transformations, which must obey two laws: the *GetPut* law, and the *PutGet* law. Intuitively, a well-behaved transformation will return identical views before and after a backward transformation using an unmodified view (*GetPut* law), and will return the same view after a backward transformation using a modified view (*PutGet* law). A lot of the work in bidirectional transformations focuses on well-behaved transformations.

Bidirectional transformations, and especially well-behaved ones, are notoriously difficult to write and maintain. Bidirectional *programming* attempts to solve that issue by providing programming languages that allow developers to define transformations in a way that is easier to write and maintain, often at the cost of a loss of expressive power [20].

Security views have been used to restrict access to data [29, 9]. Subjects may access views that will extract the data they are allowed to read from a source that also contains data that they should not have access to. These views are often not editable (except for Foster et al.'s updatable security views [11]), but they highlight the fact that views can be used as an authorisation mechanism.

We show in this paper that bidirectional programs can also be used as a form of authorisation mechanism. We show that implementing Attribute-Based Access Control (ABAC) directly using a put-based bidirectional programming language causes three types of problems:

- The laws of well-behaved transformations make it difficult to use runtime attribute values to implement ABAC using *only* bidirectional transformations.

- Including access control into a bidirectional program goes against the idea of separation of concerns.

- Including access control into bidirectional programs prevents the reuse of the same program with subjects with different access policies.

We then propose an architecture that combines put-based bidirectional programs with ABAC policies. View-centric authorisation policies are used to generate filters that sanitise views after forward transformations and before backward transformations are applied, therefore enforcing ABAC rules on the views and their corresponding sources. Those policies are written in a policy language that focuses on the specificities of bidirectional transformations over XML documents. We demonstrate our solution using a calendar sharing example.

The rest of this paper is organised as follows: in Section 2, we formally introduce bidirectional transformations, as well as the laws that govern "well-behaved" transformations. We also discuss and compare the different types of bidirectional transformation engines available. In Section 3, we use an example to illustrate the relationship between bidirectional transformations and authorisation, and highlight how put-based bidirectional programs are limited in their ability to express authorisation constraints. Section 4 is an overview of our approach to express and enforce ABAC on bidirectional programs. We present our policy language in Section 5, and authorisation filters in Section 6. In Section 7, we discuss our implementation of the approach, as well as our proof of concept to demonstrate the approach's feasability. Section 8 presents related work. We conclude this paper in Section 9, where we also discuss future work.

2. BACKGROUND

In this section, we formally introduce well-behaved bidirectional transformations, as well as different types of bidirectional programming languages.

2.1 Bidirectional transformations

A bidirectional transformation is a pair of functions that can transform a *source* into a *target*, and update the source to reflect changes made to the target [6], as illustrated on Figure 1. The *get* function produces a target from a source, in what is called a *forward transformation*. The *put* function updates the source according to changes made to the target, in what is called a *backward transformation*. The target is also often called a *view*, especially in transformations concerned with the view-update problem.

Formally [11], a bidirectional transformation is a mapping between a set of sources S and a set of views V, where we can define the *get* function as:

$$get : s \to v \tag{1}$$

and the *put* function as:

$$put : s \to v \to s \tag{2}$$

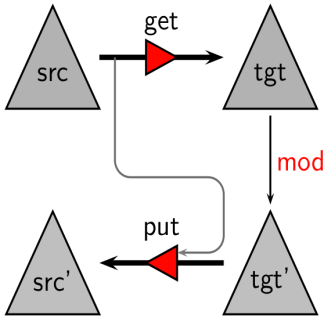


Figure 1: Get and put functions

2.2 Laws of well-formed transformations

A transformation is said to be *well-formed* when it satisfies two important laws, *GetPut* and *PutGet* [11, 20].

The *GetPut* law is the identity law. It mandates that, if a view is left unchanged since its extraction from the source (using *get*), then a backward transformation (*put*) will not alter the source. Formally, we can describe *GetPut* as:

$$put\ s\ (get\ s) = s \tag{3}$$

The *PutGet* law mandates that all changes made to the view are reflected fully to the source (during *put*), such that a subsequent *get* will preserve all the changes. Formally, we can describe *PutGet* as:

$$get\ (put\ s\ v) = v \tag{4}$$

2.3 Bidirectional programming languages

Bidirectional programming languages can be classified in two families: get-based languages and put-based languages.

2.3.1 Get-based languages

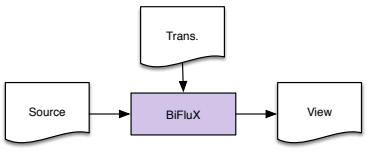
Several programming languages have been proposed that allow developers to write programs that produce bidirectional transformations. The majority of these languages are *get-based* languages, where the developer writes a *get* function, and the language's tools can derive a *put* function that, when combined with the *get* function, form a well-behaved bidirectional transformation. Examples of such languages include GRoundTram [17], Boomerang [4], and others.

However, for a given *get* function, there may be many *put* functions that would form a well-behaved bidirectional transformation. Get-based programming tools that automatically generate a *put* function given a *get* function will therefore not necessarily generate a *put* function that suits the developer's needs. A simple example illustrates the issue. We consider a rectangle, represented by its height h and width w, as the source. The *get* function returns the rectangle's height only, which is the view:

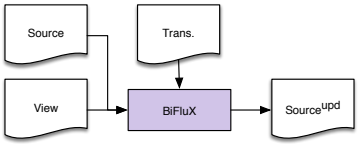
$$get(w, h) = h \tag{5}$$

Many *put* functions exist that would produce a well-behaved transformation. For example, the rectangle's original width could be used:

$$put1\ (w, h)\ h' = (w, h') \tag{6}$$



(a) Forward transformation



(b) Backward transformation

Figure 2: Bidirectional transformations in BiFluX

But many alternative *put* functions would also be acceptable, e.g.:

$$put2\ (w, h)\ h' = (w * h/h', h') \tag{7}$$

Both these *put* functions, and infinitely more, will produce a well-behaved bidirectional transformation, but not all of them will be acceptable for the developer. This is a limitation of get-based languages.

To mitigate this issue, several extensions of *get* functions have been proposed, such as quotient lenses [12], matching lenses [2], or Edit lenses [18].

2.3.2 Put-based languages

More recently, *put-based* languages (sometimes also called *putback* languages) have been proposed, such as BiFluX [27, 36]. Put-based languages ask the developer to provide a *put* function, and derive the *get* function automatically. Since it can be shown that, for a given *put* function, there is only one *get* function that produces a well-behaved bidirectional transformation [27, 36], developers have better control over the behaviour of their transformations.

BiFluX is a *putback* language for bidirectional transformations over XML files. Figure 2 shows how transformations work with BiFluX, with Figure 2a representing a forward transformation, and Figure 2b representing a backward transformation. Developers can write a program in BiFluX's language, and the compiler, implemented in Haskell [3], produces both a forward and a backward transformation, also in Haskell, that are guaranteed to form a well-behaved bidirectional transformation.

3. BIDIRECTIONAL PROGRAMMING AND ABAC

In this section, we use a calendar example to implement authorisation using bidirectional programming. We highlight and discuss the limitations of well-behaved bidirectional transformations as a way of implementing an ABAC authentication mechanism. Our example is a modified version of Foster's calendar sharing problem [11]. Appendix A shows the example in more detail. This example is a simple projection of the source into a view, chosen for its simplicity. However, bidirectional programming languages such as BiFluX can handle much more than projections, as they are able to change the structure of the data between the source and the view.

Alice maintains an online calendar, in which she records both her personal and work appointments (events). For each event, she records a start and end date and time, a location, a description, and a note. She would like to share her calendar with her colleague Bob. In order to balance her desire for privacy with Bob's need to access her calendar, she elicits the following requirements:

- Bob should be given access to Alice's work events, but not to her private events (*Req. 1*);
- Bob should only be given access to the following fields of Alice's work events: start time, end time, name, and location (*Req. 2*).

```
1  UPDATE $event IN $source/event BY
2    MATCH  ->
3      REPLACE $event/starttime WITH
           $starttime;
4      REPLACE  $event/endtime WITH
           $endtime;
5      REPLACE  $event/location WITH
           $location
6    | UNMATCHV -> CREATE VALUE
7      <event>
8        <starttime/>
9        <endtime/>
10       <name/>
11       <note>nothing</note>
12       <location/>
13       <private>False</private>
14     </event>
15   | UNMATCHS -> DELETE .
16 FOR VIEW event[$starttime AS v:
       starttime,
17   $endtime AS v:endtime, $name AS v:
       name,
18   $location AS v:location] IN $view/*
19 MATCHING SOURCE BY $event/name VIEW BY
       $name
20 WHERE private/text() = 'False'
```

Listing 1: BiFluX transformation for the calendar example

Alice's calendar is an XML document. She wants to write a bidirectional program to share her calendar with Bob, and use BiFluX to generate a bidirectional transformation from it. She writes a bidirectional program that satisfies *Reqs. 1* and *2* (Listing 1). The program features well-separated CRUD operations: the *UNMATCHV* definition, on lines 6-14, defines the *Create* permissions; the *view* definition, on lines 16-20, defines the *Read* permissions; the *MATCH* definition, on lines 2-5, defines the *Update* permissions; and finally, the *UNMATCHS* definition, on line 15, defines the *Delete* permissions.

The program satisfies Alice's requirements. Line 20 restricts the view to Alice's work events only, ensuring that Bob cannot read her personal events. Since only work events are available in the view, the *UNMATCHS* directive can only apply to work events, and therefore Bob will not be able to delete any of Alice's private events. Similarly, the *MATCH* directive guarantees that Bob will not be able to update any of Alice's private events. And finally, the *UNMATCHV* directive prescribes, on line 13, that any event

created by Bob will be added to Alice's calendar as a work event. Hence, Bob is not able to create new private events on Alice's calendar. This satisfied *Req. 1*. The view definition, on lines 16-20, stipulates which fields of Alice's events Bob can access, which satisfies *Req. 2*.

```
<?xml version="1.0"?>
<calendar>
  <event>
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <note>Prepare some slides</note>
    <location>Room 1611</location>
    <private>False</private>
  </event>
  <event>
    <starttime>2014-11-21_20:00</
        starttime>
    <endtime>2014-11-20_22:00</endtime>
    <name>Dinner</name>
    <note>Meet with Mr. Creosote</note>
    <location>Restaurant</location>
    <private>True</private>
  </event>
</calendar>
```

Listing 2: Alice's calendar

Listing 2 shows an example of Alice's calendar (the source of the transformation, which conforms to the source DTD in Appendix A.1). There are two events in the calendar, and only one of those is a work event. The forward transformation produces the view on Listing 3, which only contains the work event. The *note* field, which Bob is not authorised to see, does not appear on the view.

```
<?xml version="1.0"?>
<calview>
  <event>
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <location>Room 1611</location>
  </event>
</calview>
```

Listing 3: Bob's view of Alice's calendar (XML)

Bob can update or delete events in his view, and he can create new events, as long as his changes produce an updated view that still conforms to the view DTD (see Appendix A.2). Bob's changes can be reflected to Alice's source with a backward transformation. Listings 4 and 5 show an example. Listing 4 contains a new event that Bob added to his view. Listing 5 is the event as it is added to Alice's source as a result of the backward transformation. The fields created by Bob are added to Alice's source, as well as the default value for the *note* field. The *private* field is set to *false*, which indicates a work event. BiFluX has verified that the bidirectional transformation is well-behaved.

```
[...]
<event>
  <starttime>now</starttime>
  <endtime>later</endtime>
  <name>New Meeting</name>
  <location>The Office</location>
</event>
[...]
```

Listing 4: Addition to Bob's view

```
[...]
<event>
  <starttime>now</starttime>
  <endtime>later</endtime>
  <name>New Meeting</name>
  <location>The Office</location>
  <note>Nothing</note>
  <private>False</private>
</event>
[...]
```

Listing 5: Propagated event in Alice's source

Alice, however, is not satisfied with her program. She thinks that Bob's access to her calendar is too broad, and that her data may be at risk. To mitigate the risk she decides to incorporate a form of ABAC to her transformation. She elicits the following additional requirements:

- Bob should be able to create new or update existing events, but only during working hours[1] (*Req. 3*);
- Bob should be able to delete existing events from Alice's calendar, but only if he is at work, as determined by his IP address (*Req. 4*).

Alice wants to update her program to satisfy these requirements. In ABAC terminology, the time of the day is an environment attribute, and Bob's location is a subject attribute. For simplicity, Alice wants the value of these attributes to be evaluated when transformations are performed, as opposed to when Bob is actually making the changes. In BiFluX, Alice could simply pass the values of these two attributes to the main procedure, and use conditional statements to implement authorisation. Listing 6 shows an excerpt of the calendar program, where *Req. 3* is implemented. Alice has added, on line 2, an `if` statement, that will create a new event in the source only if the variable `$workingHours` evaluates to *true* when the transformation is run. *Req. 4* can be implemented in a similar way.

[1]We assume working hours to be Mon-Fri, 09:00am - 07:00pm

```
[...]
| UNMATCHV -> IF ($workingHours == true
    ) THEN {
  CREATE VALUE
    <event>
      <starttime/>
      <endtime/>
      <name/>
      <note>nothing</note>
      <location/>
      <private>False</private>
    </event>
} ELSE {}
[...]
```

Listing 6: Updated transformation (portion)

Unfortunately, the transformation produced from Alice's modified program does not satisfy *PutGet* for some combination of the attributes' values. For example, if Bob creates a new event (Listing 4) during his working hours, the source will be updated just like in Alice's initial program (Listing 5). Any new forward transformation will produce a source with the added event, which will be identical to Bob's updated view. This is *PutGet*. However, if the backward transformation happens *outside* of Bob's working hours, then the new event will *not* be added to Alice's source, and any subsequent forward transformation will not include the event, and will therefore be different from Bob's view, which violates *PutGet*. *GetPut*, however, is still satisfied since it involves no changes in the view, but the introduction of rules based on the values of attributes to govern what Bob can read will cause the same issue.

The issue is more easily highlighted by considering a very simple bidirectional transformation in Haskell, that could have been produced by a simple BiFluX program. The following *get* function takes a list as its source, and returns the first element of the list as the view. The *put* function replaces the first element of the source with its single element:

```
get :: Source -> View
get (x:xs) = x

put :: Source -> View -> Source
put y (x:xs) = y:xs
```

It is obvious that both *GetPut* and *PutGet* hold. We then refine the two functions, so that the *get* function will only return the first element of the source list if an attribute a evaluates to true, and an empty list otherwise. Similarly, the *put* function will only update the source with its view element if a evaluates to true, and leave the source otherwise unchanged:

```
get :: Bool -> Source -> View
get a (x:xs) = if a then x
               else []

put :: Bool -> Source -> View -> Source
put a y (x:xs) = if a then y:xs
                 else x:xs
```

In this case, *GetPut* and *PutGet* only hold if the value of a does not change between operations. This is a significant issue for our example, as it would force Bob to get a new view right before updating Alice's calendar every time he wants to update it. Another issue arises if we modify the functions a bit more. Now, *get* is unchanged, but *put* takes b instead of a as an argument. The functions become:

```
get :: Bool -> Source -> View
get a (x:xs) = if a then x
               else []

put :: Bool -> Source -> View -> Source
put b y (x:xs) = if b then y:xs
                 else x:xs
```

If the values of a and b are independent, then it is not possible to guarantee that *GetPut* and *PutGet* hold anymore, even if their respective values are fixed. *GetPut* is then expressed as:

put b s (get a s) = s

PutGet is then expressed as:

get a (put b s v) = v

Let us consider a to be true and b false. Given the list `[1,2]` as a source, and considering that we update the view, when appropriate, with the value 4, Table 1 shows whether *GetPut* and *PutGet* hold for each value of a and b.

The second line, where a is true and b is false, can be detailed as follows. For *GetPut*, we first run `get true [1,2]`, which returns `1`. We then run `put false [1,2] 1`, which returns `[1,2]`. *GetPut* holds. For *PutGet*, we first run `put false [1,2] 4`, which returns `[1,2]`. We then run `get true [1,2]`, which returns `1`. *PutGet* does not hold. The other combinations of values of a and b can be developed in the same way.

This example highlights the limitation to the expressivity of authorisation rules implemented as part of a bidirectional transformation. If a transformation uses attributes whose value may vary, then the transformation will not be well-behaved for some combinations of the attributes' values.

Another issue with this solution is that the authorisation code (*what* can be done, under which conditions) is mixed with the implementation (*how* it can be done), which violates the principle of separation of concerns [7].

Finally, a third issue has to do with reuse and maintainability. It is possible that Alice may want to share the same data with more people than just Bob, but with slightly different requirements. For example, she may want to give read-only access to her work-related events to some of her subordinates, or give write access to somebody else, but only for events related to a particular project. If Alice integrates the authorisation constraints directly into the program, she will have to write a program for each of those people that she wants to share a view with. If the structure of Alice's calendar, or of the views, must change, then she will have to edit many transformations.

In the next section, we propose an approach that solves these three issues. A separation of the authorisation rules from the program achieves separation of concerns, and also allows one to define well-behaved transformations that BiFluX can run.

4. OVERVIEW OF THE APPROACH

To address the issues discussed above, we propose an approach that separates the expression and enforcement of the

Table 1: Status of *GetPut* and *PutGet* for values of a and b

a	b	GetPut	Holds?	PutGet	Holds?
T	T	`put true [1,2] (get true [1,2]) = [1,2]`	Y	`get true (put true [1,2] 4) = 4`	Y
T	F	`put false [1,2] (get true [1,2]) = [1,2]`	Y	`get true (put false [1,2] 4) = 1`	N
F	T	`put true [1,2] (get false [1,2]) = [[],2]`	N	`get false (put true [1,2] 4) = []`	N
F	F	`put false [1,2] (get false [1,2]) = [1,2]`	Y	`get false (put false [1,2] 4) = []`	N

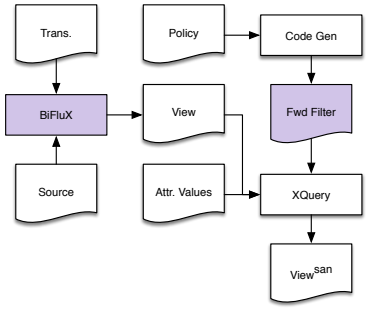


Figure 3: Forward transformation, with authorisation filter

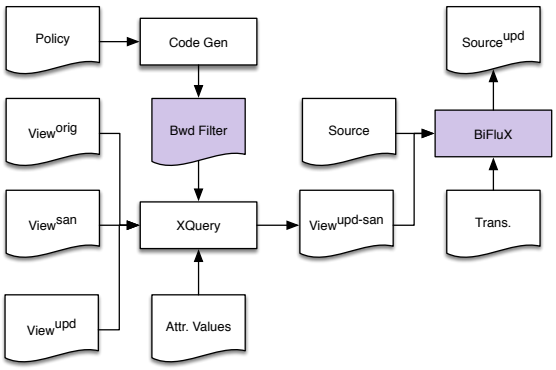


Figure 4: Backward transformation, with authorisation filter

authorisation policy from the program. In our approach, the program is written without consideration for authorisation attributes: all changes to the view (creation, update and deletion of elements) are authorised, and the view contains all the information that could ever be read. The part of the program that defines the view acts as a "best scenario" *read* policy, while the *MATCH*, *UNMATCHS* and *UNMATCHV* parts of the program define *how* the update, delete and create operations are carried on, respectively, if they were to be authorised. The program must produce a well-behaved bidirectional transformation. An authorisation policy is written separately, to specify the conditions under which create, read, update and delete operations can be reflected to the source. The policy is then used to produce two filters, one for forward transformations, and one for backward transformations. Figures 3 and 4 show how the filters are used in conjunction with BiFluX for the generation of the view (forward transformation), and the propagation to the source (backward transformation), respectively.

The policy (*Policy*) is first compiled into a forward filter (*Fwd Filter* in Figure 3) and a backward filter (*Bwd Filter* in Figure 4).

To get the view from the source (Figure 3), the forward transformation is run first. It uses the source (*Source*) and produces a view ($View$). Then, the view is passed through the forward filter ($Fwd\ Filter$) using the current values of the attributes involved ($Attr.\ Values$), which produces a sanitised view ($View^{san}$), which can be shared with the recipient (e.g., Bob).

To reflect the changes made to the view back into the source (Figure 4), the view originally produced ($View^{orig}$), the view shared with the recipient after the forward filter ($View^{san}$) and the view as updated by the recipient ($View^{upd}$) are passed through the backward filter ($Bwd\ Filter$), together with the current values of the attributes involved ($Attr.\ Values$). Any change that is not allowed at that moment will be reverted by the backward filter. Alternatively, one could reject the update entirely if some unauthorised changes are detected. The resulting view ($View^{upd-san}$) is then passed, together with the original source (*Source*), to BiFluX to run the backward transformation. The result is an updated source ($Source^{upd}$), that has only been updated with the authorised changes.

This approach separates the authorisation rules from the program. It allows for a clearer expression of the authorisation policy, as well as a clearer expression of the program. The separation also allows one to define complex, attribute-based authorisation policies that would otherwise result in transformations that are not well-behaved.

Because the policy is defined over a *view*, the same program can be reused to share views with multiple subjects, by defining a different policy for each of them, or each group of them that can be governed by the same policy.

Our approach applies to simple projections of a source onto a view (Alice's calendar sharing requirements define an example of a simple projection), where bidirectional transformations are a way of addressing the view-update problem. But it also applies to more complex transformations where the structures of the source and view are different. For example, a source for an online store may contain a set of products, each of then containing stock information and the details of all orders of the product; while the view would be a set of orders from customers, each order containing details about the products ordered. In both cases, the authorisation policy will determine which parts of the view can be created, read, updated or deleted.

5. A POLICY LANGUAGE FOR BIDIRECTIONAL TRANSFORMATIONS

BXauthZ is a simple policy language to express attribute-based rules on XML views. BXauthZ allows one to define rules for CRUD operations. The resources on which rules apply are defined as XPath expressions [34]. The language

⟨*Policy*⟩ ::= '`policy`' ⟨*id*⟩ '`{`' ⟨*subject*⟩ ⟨*transformation*⟩ ⟨*rule*⟩ '`}`'

⟨*Subject*⟩ ::= '`subject`' ⟨*id*⟩

⟨*Transformation*⟩ ::= '`transformation`' ⟨*id*⟩

⟨*Rules*⟩ ::= ⟨*Rule*⟩ ⟨*Rules*⟩
| ⟨*Rule*⟩

⟨*Rule*⟩ ::= '`rule`' ⟨*id*⟩ '`{`' ⟨*Actions*⟩ ⟨*Resources*⟩ ⟨*Conditions*⟩ '`}`'
| '`rule`' ⟨*id*⟩ '`{`' ⟨*Actions*⟩ ⟨*Resources*⟩ '`}`'

⟨*Actions*⟩ ::= ⟨*Action*⟩ ⟨*Actions*⟩
| ⟨*Action*⟩

⟨*Resources*⟩ ::= ⟨*Resource*⟩ ⟨*Resources*⟩
| ⟨*Resource*⟩

⟨*Conditions*⟩ ::= ⟨*Condition*⟩ ⟨*Conditions*⟩
| ⟨*Condition*⟩

⟨*Action*⟩ ::= '`action`' '`create`'
| '`action`' '`read`'
| '`action`' '`update`'
| '`action`' '`delete`'

⟨*Resource*⟩ ::= '`resource`' ⟨*XPathExpression*⟩
| '`resource`' ⟨*XPathExpression*⟩ '`matching-by`' ⟨*Name*⟩

⟨*Condition*⟩ ::= '`condition`' ⟨*BooleanExpression*⟩

Figure 5: Grammar of BXauthZ (abbreviated)

is loosely based on Axiomatics' ALFA syntax[2], which is a concise DSL to write policies that can be compiled into eXtensible Access Control Markup Language (XACML) [26] (note that BXauthZ does not compile to XACML).

5.1 Grammar

Figure 5 shows an abbreviated version of BXauthZ's grammar. A policy in BXauthZ describes authorisation permissions for one or several subjects to perform CRUD operations over XML data in a view. A policy has a unique name, and includes the subjects and the transformation that produces the view on which the policy applies (both mandatory), as well as a set of rules (optional, though an empty set of rules would not produce a very useful policy). Any action on any resource is forbidden, unless explicitly permitted by a rule.

Each rule has a unique name, and is made of three parts: a set of actions (mandatory), a set of resources (mandatory), and a set of conditions (optional). Each rule's effect is *Permit*, as long as *all* the conditions are satisfied. The actions can be *create*, *read*, *update* or *delete*. The resources are expressed as XPath expressions [34]. If the resource is an XML node, then the permission applies to the node as well as all its contents, including descendant nodes. Conditions are boolean expression. Resources for *create*, *update*, and *delete* actions also have a `matching-by` statement which denotes the child element or attribute that is used to match elements, in order to tell the difference between an update and a creation or deletion of an element.

There is no rule combination algorithm in BXauthZ, unlike policy languages such as XACML. Since all the rules define *Permit* effects, and since anything that is not explicitly permitted is denied, there can not be any conflicting answers, and therefore the order in which the rules are evaluated does not matter. If several rules use the same XPath expression, then the satisfaction of any of the rules will allow for the action defined by that rule.

5.2 Example

Listing 7 shows a sample policy written by Alice, to regulate Bob's access to the view generated by the program Alice created on Listing 1. The policy conforms to the 4 requirements elicited by Alice in Section 3.

```
1  policy BobCalendar {
2          subjects {Bob}
3          transformation calendar
4          rule CalRead {
5                  action read
6                  resource /calview/*
7          }
8          rule CalCreate {
9                  action create
10                 action update
11                 resource /calview/*
                       matching-by name/
                       text()
12                 condition $workingHours
13         }
14         rule CalUpdateDelete {
15                 action delete
16                 resource /calview/*
                       matching-by name/
                       text()
17                 condition $atOffice
18         }
19 }
```

Listing 7: Policy for Bob's access to Alice's calendar

The policy's only subject is Bob (line 2), and the policy applies to views produced by the program on Listing 1 (line 3). There are four rules in the policy. The first one (lines 4-7) defines the read actions (line 5). Bob is allowed to read everything, at all times (line 6). Therefore, the forward filter generated by this policy will leave the generated view intact. Because the view definition in the program already excludes private events from the view, and because it already restricts the elements of each events that can be accessed, *Reqs. 1* and *2* are satisfied.

The second rule (lines 8-13) has two actions, create (line 9) and update (line 10). The resources are all the events in the view (line 11). This rule has a condition (line 12), which states that the rule only applies if `$workingHours` evaluates to *true*, which will only be the case during working hours, as defined by Alice. This rule satisfies *Req. 3*.

The third rule (lines 14-18) has only one action, delete (line 15). The resources are all the events in the view (line 16), like the other two rules. The rule has one condition (line

[2] `http://www.axiomatics.com/alfa-plugin-for-eclipse.html`, accessed January 2015

17), which states that the rule only applies if `$atOffice` evaluates to *true*, which will only be the case if Bob's IP address shows that he is located on the company premises. The rule satisfies *Req. 4*.

5.3 Remark

Nodes that are not covered by any rule, and who do not have at least one ancestor covered by a rule, will not always be removed from the view. If such a node has at least one descendant covered by a rule, then the node will be conserved in the filtered view, but without its content, except for the descendants that must be conserved as well. Let us consider this simple view as an example:

```
<a>
  <b/>
  <c>
    <d/>
  </c>
</a>
```

There could be a policy that applies to that view, and defines only one rule, with `/a/c/d` as a resource. While the `a` and `c` elements are not covered by the rule, they are ancestors of `d`, which is captured by the rule. `b`, however, is not an ancestor of `d`. Therefore, the sanitised view that would result from the application of the policy would be:

```
<a>
  <c>
    <d/>
  </c>
</a>
```

6. AUTHORISATION FILTERS

The BXauthZ compiler produces, for each policy, a pair of XQuery filters. The forward filter, which is run after the forward transformation, sanitises the view according to the *read* actions in the policy, so it can be shared with its recipient. The backward filter, which is run before the backward transformation, sanitises the view according to the *create*, *update* and *delete* actions in the policy, to guarantee that no unauthorised changes are propagated to the source. Both filters will use the values of the attributes defined in the rules' conditions. Those values should be obtained securely, for example through a Policy Information Point (PIP). Determining which values can be collected from the client (e.g. Bob) without verification is out of the scope of this paper.

6.1 The forward filter

The forward filter takes two inputs: the view generated by the forward transformation, and the values of the attributes involved in the filter. The filter outputs a sanitised view. We call the view generated by the forward transformation, $view^{orig}$, and the sanitised view produced by the forward filter, $view^{sanV}$. Using the XPath expressions defined in the policy, the filter removes from the view the elements to which access is not granted at the time the filter is run.

6.2 The backward filter

The backward filter takes three inputs. The first one is the view updated by the user, which we call $view^{upd}$. The second one is the original view, $view^{orig}$. The third one is the view produced by the forward filter, $view^{sanV}$. The filter outputs a sanitised view that we call $view^{sanS}$. The filter also takes as an input the values of all the attributes involved in *create*, *update*, and *delete* decisions.

This backward filter is more complex than the forward filter. Indeed, it needs to merge the changes made by the user, as well as the elements that have been hidden from the user by the forward filter. First, the filter uses $view^{sanV}$, $view^{sanS}$, and the attributes' values to revert the changes made to the view that are not permitted by the policy. Then, the filter uses this product together with $view^{orig}$ to add the elements that had been remove by the forward filter. The resulting view can them be used by the backward transformation to reflect the user's changes back to the source.

7. IMPLEMENTATION AND PROOF OF CONCEPT

Our implementation of the approach described in this paper, as well as the calendar example, are available online.

7.1 Policy language and filter generation

BXauthZ[3] has been implemented as a Domain-Specific Language (DSL) using Xtext[4], an Eclipse-based framework for developing DSLs and programming languages. BXauthZ offers a complete IDE based on eclipse, as well as code generation capabilities that generate both the forward and the backward filters for any policy.

7.2 Filter evaluation

The filters are generated by BXauthZ in XQuery 3.0 [33]. Any product that complies with the XQuery 3.0 recommendation should be able to execute the filters. To conduct our tests, we used Zorba 3.0[5], an open source, multi-platform XQuery and JSON query processor.

7.3 Proof of concept

To evaluate our approach, we have developed a proof of concept based around the calendar example used throughout this paper. Bob's policy, as well as a few alternatives, were created using BXauthZ, and the corresponding filters were generated using BXauthZ's code generator[6].

8. RELATED WORK

Secure XML views [29] have been studied in detail as a means to provide access to confidential information. Fan et al. have proposed an approach to support XPath queries over security views [9]. Kuper et al. generalised the notion of security views where authorisation policies are specified over DTDs [23]. Rota et al. integrate XACML with OWL [31] ontologies to provide semantic authorisation for XML documents [28].

As far as we are aware, the only work that addresses the view-update problem in security views is Foster's updatable security views [11, 10], for which there is no implementation. Foster introduces secure lenses as an extension of his previous work on lenses [13], with a type system to ensure integrity and confidentiality of the data in the source [11]. While the calendar example in this paper is inspired by Foster's, secure lenses are very different from the solution we

[3] `https://github.com/lmontrieux/bxauthz`
[4] `http://www.xtext.org`
[5] `http://www.zorba.io`
[6] `https://github.com/lmontrieux/biflux-filters-poc`

propose, in the sense that he uses annotations on the source to enforce the confidentiality and integrity of some of the source data, while we devise policies on the view to ensure confidentiality and integrity, and implement them around a bidirectional transformation engine, rather than extending its semantics to support it, which Foster does (although with Boomerang [4] instead of BiFluX).

Access control for XML documents is also a field that has been widely researched. For example, Kudo and Hada proposed XML Access Control Language (XACL), a language that provides authorisation for XML documents, based on a provisional authorisation model, and using XPath expressions [21, 22]. Gabillon and Bruno transform authorisation policies into XSLT sheets [32], that are used to extract a secure view from XML documents [14]. Auntariya et al. propose a rule-based access control model that provides declarative policy rules on XML documents, as well as conflict resolution and default authorisation [1]. Gowadia and Farkas use RDF statements with authorisation properties to represent XML access control rules [15]. Zhang et al. use XML Schemas to represent Role-Based Access Control (RBAC) models for XML data [37]. Byun and Park introduce a two phase filtering approach to modify queries on XML databases in order to ensure that the results will not violate access control policies [5]. Duong and Zhang describe an access control model for XML that supports both read and write authorisation, allowing authorised users to change the structure of the XML documents [8]. Finally, Thimma et al. introduce Hybrid XML Access Control (HyXAC), a hybrid access control approach that provides secure queries on XML documents while improving its performance over other solutions [30].

9. CONCLUSION

In this paper, we highlighted the strong connection between authorisation and bidirectional transformations, as transformations themselves can provide a simple form of access control. We provided a solution that allows for the enforcement of attribute-based authorisation on bidirectional programs, without compromising on the laws of well-behaved bidirectional transformations. Our approach uses a custom policy language that is used to generate filters that sanitise views after a forward transformation and before a backward transformation. The former guarantees that no unauthorised data is leaked to the recipient of the view, while the latter guarantees the integrity of the source. Our approach enforces a clear separation of concerns between the programs and the authorisation policies, therefore allowing one to reuse the same transformation to share information with several subjects, simply by creating a different policy for each subject or group of subjects.

A first direction for future work would be to derive the entire program from the policy alone, which would allow users such as Alice to share data in what we hope would be a much easier way, while still conserving the advantages of bidirectional transformations. Our experience shows that writing non-trivial, well-behaved programs can be very challenging. However, such a solution would reduce the user's ability to control the update, and in particular may restrict the view to a projection of the source.

Another direction for future work is the exploration of other bidirectional transformation engines, in particular get-based solutions. Because get-based solutions do not require the user to explicitly define a put strategy, it is the engine itself that selects a strategy that, given a forward transformation, will satisfy both the *PutGet* and *GetPut* laws. If authorisation is considered in the approach, the choice of a *put* function should satisfy the authorisation policy.

Yet another direction is the implementation of ABAC constructs directly into bidirectional transformation languages, with the goal of making the transformations more efficient. This may require us to relax the laws of well-behaved transformations in order to allow for attribute-based authorisation, while still offering strong guarantees that transformations behave as expected.

Finally, issues of conflicts and performance should be studied. Sharing data using multiple views, that are shared to different subjects, will lead to conflicts that will need to be resolved. For example, if a view deletes a node whilst another one modifies it, a conflict resolution strategy will be necessary. Performance should also be considered. Efficient bidirectional transformation engines will allow for practical use over large documents.

10. ACKNOWLEDGEMENTS

This work is supported financially by the Nation Basic Research Program (973 Program) of China (grant No. 2015CB352201) and by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 in Japan.

11. REFERENCES

[1] C. Anutariya, S. Chatvichienchai, M. Iwiahara, V. Wuwongse, and Y. Kambayashi. A rule-based XML access control model. In M. Schröder and G. Wagner, editors, *Rules and Rule Markup Languages for the Semantic Web*, number 2876 in Lecture Notes in Computer Science, pages 35–48. Springer, 2003.

[2] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[3] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, London; New York, 2nd edition, May 1998.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. ACM.

[5] C. Byun and S. Park. Two phase filtering for XML access control. In W. Jonker and M. Petković, editors, *Secure Data Management*, number 4165 in Lecture Notes in Computer Science, pages 115–130. Springer, Jan. 2006.

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, number 5563 in Lecture Notes in Computer Science, pages 260–283. Springer, Jan. 2009.

[7] P. D. E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer, 1982.

[8] M. Duong and Y. Zhang. An integrated access control for securely querying and updating XML data. In A. Fekete and X. Lin, editors, *Nineteenth Australasian*

Database Conference (ADC 2008), volume 75 of *CRPIT*, pages 75–83, Wollongong, NSW, Australia, 2008. ACS.

[9] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 587–598, New York, NY, USA, 2004. ACM.

[10] J. Foster, B. Pierce, and S. Zdancewic. Updatable security views. In *22nd IEEE Computer Security Foundations Symposium, 2009. CSF '09*, pages 60–74, July 2009.

[11] J. N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pensylvania, Dec. 2009.

[12] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and provenance. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 271–280, New York, NY, USA, 2008. ACM.

[13] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 233–246, New York, NY, USA, 2005. ACM.

[14] A. Gabillon and E. Bruno. Regulating access to XML documents. In M. S. Olivier and D. L. Spooner, editors, *Database and Application Security XV*, number 87 in IFIP — The International Federation for Information Processing, pages 299–314. Springer, Jan. 2002.

[15] V. Gowadia and C. Farkas. RDF metadata for XML access control. In *Proceedings of the 2003 ACM Workshop on XML Security*, XMLSEC '03, pages 39–48, New York, NY, USA, 2003. ACM.

[16] F. Hermann and J. Voigtländer. First international workshop on bidirectional transformations (BX 2012): Preface. *Electronic Communications of the EASST*, 49(0), July 2012.

[17] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 205–216, New York, NY, USA, 2010. ACM.

[18] M. Hofmann, B. Pierce, and D. Wagner. Edit lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 495–508, New York, NY, USA, 2012. ACM.

[19] B. Hoisl, Z. Hu, and S. Hidaka. Towards co-evolution in model-driven development via bidirectional higher-order transformation. pages 466–471, Jan. 2014.

[20] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods*, number 8442 in Lecture Notes in Computer Science, pages 1–15. Springer, Jan. 2014.

[21] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 87–96, New York, NY, USA, 2000. ACM.

[22] M. Kudo and N. Qi. Access control policy models for XML. In T. Yu and S. Jajodia, editors, *Secure Data Management in Decentralized Systems*, number 33 in Advances in Information Security, pages 97–126. Springer, Jan. 2007.

[23] G. Kuper, F. Massacci, and N. Rassadko. Generalized XML security views. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 77–84, New York, NY, USA, 2005. ACM.

[24] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 146–155, New York, NY, USA, 2007. ACM.

[25] K. Nakano, Z. Hu, and M. Takeichi. Consistent web site updating based on bidirectional transformation. *International Journal on Software Tools for Technology Transfer*, 11(6):453–468, Dec. 2009.

[26] OASIS. eXtensible access control markup language (XACML) version 3.0, Jan. 2013.

[27] H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *6th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, 2014.

[28] A. Rota, S. Short, and M. A. Rahaman. XML secure views using semantic access control. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 5:1–5:10, New York, NY, USA, 2010. ACM.

[29] A. Stoica and C. Farkas. Secure XML views. In E. Gudes and S. Shenoi, editors, *Research Directions in Data and Applications Security*, number 128 in IFIP — The International Federation for Information Processing, pages 133–146. Springer, 2003.

[30] M. Thimma, T. K. Tsui, and B. Luo. HyXAC: A hybrid approach for XML access control. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, pages 113–124, New York, NY, USA, 2013. ACM.

[31] W3C. OWL web ontology language reference, Feb. 2004.

[32] W3C. XSL transformations (XSLT) version 2.0, Jan. 2007.

[33] W3C. XML XPath language (XPath) 3.0, Apr. 2014.

[34] W3C. XQuery 3.0: An XML query language, Apr. 2014.

[35] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 540–550, June 2012.

[36] T. Zan, H. Pacheco, and Z. Hu. Writing bidirectional model transformations as intentional updates. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 488–491, New York, NY, USA, 2014. ACM.

[37] X. Zhang, J. Park, and R. Sandhu. Schema based XML security: RBAC approach. In S. D. C. d. Vimercati, I. Ray, and I. Ray, editors, *Data and Applications Security XVII*, number 142 in IFIP International Federation for Information Processing, pages 330–343. Springer, 2004.

APPENDIX

A. CALENDAR SHARING EXAMPLE

Due to space constraints, the calendar sharing example in this paper had to be kept short. We present an expanded version in this section.

A.1 Source DTD

This is the DTD to which the source file, i.e. Alice's entire calendar, has to conform.

```
<!DOCTYPE calendar [
  <!ELEMENT calendar (event*)>
  <!ELEMENT event(starttime, endtime,
      name, note,
    location, private)>
  <!ELEMENT starttime (#PCDATA)>
  <!ELEMENT endtime (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT note (#PCDATA)>
  <!ELEMENT location (#PCDATA)>
  <!ELEMENT private (#PCDATA)>
]>
```

A.2 View DTD

This is the DTD to which the view file, i.e. Bob's view of Alice's calendar, has to conform.

```
<!DOCTYPE calview [
  <!ELEMENT calview (event*)>
  <!ELEMENT event (starttime, endtime,
      name,
    location)>
  <!ELEMENT starttime (#PCDATA)>
  <!ELEMENT endtime (#PCDATA)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT location (#PCDATA)>
]>
```

A.3 Transformation without Access Control

This is the transformation written by Alice to share a view of her calendar with Bob. The transformation mandates how changes to Bob's view must be reflected to Alice's source. It also shows that only private events will appear in Bob's view, and that the `note` field will be hidden.

```
UPDATE $event IN $source/event BY
  MATCH  ->
    REPLACE $event/starttime WITH
        $starttime;
    REPLACE  $event/endtime WITH
        $endtime;
    REPLACE  $event/location WITH
        $location
  | UNMATCHV -> CREATE VALUE
    <event>
      <starttime/>
      <endtime/>
      <name/>
      <note>nothing</note>
      <location/>
      <private>False</private>
    </event>
  | UNMATCHS -> DELETE .
FOR VIEW event[$starttime AS v:
    starttime,
  $endtime AS v:endtime, $name AS v:
      name,
  $location AS v:location] IN $view/*
MATCHING SOURCE BY $event/name VIEW BY
    $name
WHERE private/text() = 'False'
```

A.4 Policy

This is the policy written by Alice to further restrict what Bob is allowed to do with his view of her calendar.

```
policy BobCalendar {
        subjects {Bob}
        transformation calendar
        rule CalRead {
                action read
                resource /calview/*
        }
        rule CalCreate {
                action create
                action update
                resource /calview/*
                    matching-by name/
                    text()
                condition $workingHours
        }
        rule CalUpdateDelete {
                action delete
                resource /calview/*
                    matching-by name/
                    text()
                condition $atOffice
        }
}
```

A.5 Source XML

This is Alice's calendar at the beginning of our example. This XML file conforms to the DTD in Section A.1.

```
<?xml version="1.0"?>
<calendar>
  <event>
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <note>Prepare some slides</note>
    <location>Room 1611</location>
    <private>False</private>
  </event>
  <event>
    <starttime>2014-11-21_20:00</
        starttime>
    <endtime>2014-11-20_22:00</endtime>
    <name>Dinner</name>
    <note>Meet with Mr. Creosote</note>
    <location>Restaurant</location>
    <private>True</private>
  </event>
</calendar>
```

A.6 View XML

This is Bob's view of Alice's calendar after a forward transformation, using the transformation in Section A.3. Since the access control policy in Section A.4 allows for Bob to read the entire view under any circumstances, this view is also the view obtained after the forward filter.

```
<?xml version="1.0"?>
<calview>
  <event>
```

```
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <location>Room 1611</location>
  </event>
</calview>
```

A.7 Updated view

This is Bob's view after he has modified it. Bob has added a new event, and deleted the existing one. The new event can only be reflected in Alice's source during working hours, while the deleted event can only be removed from Alice's source when Bob is at the office.

```
<?xml version="1.0"?>
<calview>
  <event>
    <starttime>2015-02-11_16:00</
        starttime>
    <endtime>2015-02-11_18:00</endtime>
    <name>Performance review</name>
    <location>Room 2005</location>
  </event>
</calview>
```

A.8 Updated views, after backward transformation

The view in the previous section is then passed to the backward filter. Depending on the time of the day and Bob's location, the resulting, sanitised view could take one of four forms. The first possible sanitised view results from running the backward transformation during working hours, and while Bob is at the office. In this case, the sanitised view is identical to the view in Section A.7.

Another possibility, if the backward filter is run during working hours but while Bob is not at the office, is that the newly created event is still present, but the deleted event has been reinstated in the view:

```
<?xml version="1.0"?>
<calview>
  <event>
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <location>Room 1611</location>
  </event>
  <event>
    <starttime>2015-02-11_16:00</
        starttime>
    <endtime>2015-02-11_18:00</endtime>
    <name>Performance review</name>
    <location>Room 2005</location>
  </event>
</calview>
```

The other two possible views are omitted due to space constraints.

A.9 Updated sources

Once the backward filter has been run, BiFluX can then safely reflect the changes to the view back to the source. The previous section showed to possible sanitised views. The first one, where both changes made by Bob were accepted, will produce the following updated source:

```
<?xml version="1.0"?>
<calendar>
  <event>
    <starttime>2015-02-11_16:00</
        starttime>
    <endtime>2015-02-11_18:00</endtime>
    <name>Performance review</name>
        <note>Nothing</note>
    <location>Room 2005</location>
    <private>False</private>
  </event>
  <event>
    <starttime>2014-11-21_20:00</
        starttime>
    <endtime>2014-11-20_22:00</endtime>
    <name>Dinner</name>
    <note>Meet with Mr. Creosote</note>
    <location>Restaurant</location>
    <private>True</private>
  </event>
</calendar>
```

The second one, where only the newly created event was accepted, will produce the following updated source:

```
<?xml version="1.0"?>
<calendar>
  <event>
    <starttime>2014-11-20_14:00</
        starttime>
    <endtime>2014-11-20_15:00</endtime>
    <name>Group meeting</name>
    <note>Prepare some slides</note>
    <location>Room 1611</location>
    <private>False</private>
  </event>
  <event>
    <starttime>2014-11-21_20:00</
        starttime>
    <endtime>2014-11-20_22:00</endtime>
    <name>Dinner</name>
    <note>Meet with Mr. Creosote</note>
    <location>Restaurant</location>
    <private>True</private>
  </event>
  <event>
      <starttime>2015-02-11_16:00</
          starttime>
    <endtime>2015-02-11_18:00</endtime>
    <name>Performance review</name>
        <note>Nothing</note>
    <location>Room 2005</location>
        <private>False</private>
  </event>
</calendar>
```