2011 IEEE 13th International Symposium on High-Assurance Systems Engineering

Conflict Detection and Lifecycle Management for Access Control in Publish/Subscribe Systems

Patrick Hein
BearingPoint, Frankfurt, Germany

Debmalya Biswas
Nokia Research, Lausanne, Switzerland

Leonardo A. Martucci, Max Mühlhäuser
Technische Universität Darmstadt, Germany

***Abstract*—In today's collaborative business environment there is a need to share information across organizational boundaries. Publish/Subscribe systems are ideal for such scenarios as they allow real-time information to be shared in an asynchronous fashion. In this work, we focus on the access control aspect. While access control in general for publish/subscribe systems has been studied before, their usage in a multi-organizational scenario leads to some novel challenges. Here a publisher might wish to enforce restrictions w.r.t. not only subscribers, but also other publishers publishing certain event types due to competitive or regulatory reasons. With different publishers and subscribers having their own preferences and restrictions, conflicts are evident w.r.t. both publishing and subscribing to specific event types. Given this, the first contribution of this work is to provide efficient conflict detection and resolution algorithms The other important (and often ignored) aspect of large scale and evolving systems is that of efficiently handling modifications to existing policies, e.g. a rule may become invalid after a certain period of time. Our approach in handling such modifications is two-fold: (i) to maintain consistency and (ii) to automatically detect and enforce rules which could not have been enforced earlier due to conflicts. The second contribution of our work is thus to provide lifecycle management for access control rules, which is tightly coupled with the conflict detection and resolution algorithms.**

***Index Terms*—Publish/subscribe systems; Access control policies; Conflict detect and resolution; Lifecycle management;**

I. Introduction

In today's collaborative business environment there is a growing need to share information across organizational boundaries. Instances of such scenarios include large scale supply chains, e.g. automotive, manufacturing, where component development is outsourced to different organizations who then collaborate to construct and market the final product. Another scenario for a collaborative business environment is an e-Services marketplace where different services may be provided by different providers, with the possibility to compose new services out of already existing ones. Publish/-Subscribe systems [1] are ideal for such large-scale distributed applications. They follow a loosely coupled architecture that allows interaction among heterogeneous applications hosted by different organizations.

Access control in publish/subscribe systems is an open research topic. While there are few research works that have considered access control models specifically for publish/-subscribe systems [2], [3], [4], most of them are inherently centralized and not applicable in a multi-organizational scenario. In other words, they are not space independent as the publishers and subscribers need to know each other. For a detailed discussion of related works, the interested reader is referred to Section VII.

Access control in multi-organizational scenarios leads to some novel challenges, not addressed in the literature (to the best of our knowledge). With multiple parties involved, each organization has its own preferences w.r.t. to who can or cannot publish (subscribe to) events of specific types. For instance, a publisher P may wish to be the sole publisher of events of type X. Such a preference by P implies restrictions on other publishers Q also interested in publishing events of type X. In other words, the preferences of different organizations may lead to conflicts. The underlying reason behind such conflicts may be simple economics or legal regulations the publishers/subscribers have to conform to. The conflicts may further be generalized to conflicts w.r.t. events of 'related' types. For instance, P's preference to be the sole publisher of events of type X may also affect publishers of related event types Y. In our work, we consider a specific relationship, that of hierarchically related event types. We consider a hierarchical ordering among the event types, so that a preference or restriction w.r.t. an event of type X also has implications on ancestors/descendents of X in the hierarchical event model. An orthogonal challenge here is the maintenance of such systems. Large scale collaborations are usually long term in the range of several years, and keep evolving over time. Evolution refers to publishers/subscribers leaving or joining the ecosystem, or even specific rules becoming invalid after a certain period of time. We refer to this aspect as lifecycle management. We outline a couple of motivational scenarios in the sequel to highlight the above issues in real-life systems.

A. Use Cases

1) Scenario A - Downstream: In scenario A, we consider a group of companies who provide content for a TV broadcaster. The TV broadcaster is categorizing content for its customers in different channels (Fig. 1). In this scenario, publishers are content providers who provide TV content for subscribers. Subscribers are customers who can subscribe to different channels. Publishers may be interested in restricting access to their content because they charge subscription fees or they have to enforce age restrictions for some movies. Subscribers can be described by different profiles e.g. $premium\_user$/$normal\_user$ or age, for instance. These profiles can be used to specify an access control for the publisher's content types.

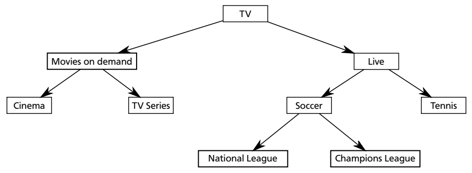

Fig. 1. TV - downstream Scenario

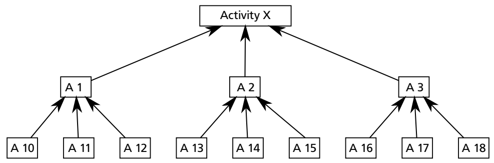

Fig. 2. Workflow - upstream scenario

$subscribe(soccer) = premium\_users$
$subscribe(movies) = premium\_users \wedge age \geq 18$

To offer a common package of all soccer games, it is useful for the TV broadcaster to allow subscribers who are permitted for soccer to also have access to the subcategories $national\ league$ and $champions\ league$. We call this way of inheritance downstream. Downstream inheritance implies that a subscriber having access to events of category X, also has access to all subcategories of X.

A publisher-publisher conflict between two or more publishers for the $soccer$ channel occurs if one of them is offering its content only to $premium\_users$ and the other to all subscribers. A publisher-publisher conflict w.r.t. hierarchically related events might be if one publisher is offering its $soccer$ content to all subscribers and another is offering its content for $national\ league$ only to $premium\_users$. There are clearly numerous ways of resolving such conflicts. An option to resolve the above conflict would be for all involved publishers to start publishing $soccer$ only to $premium\_users$.

We next discuss the 'evolving' aspect. If the conflicting publisher who wanted to restrict access of $soccer$ only to $premium\_users$ goes offline, then the previous rule of publishing $soccer$ to all subscribers can be made active again. An instance of evolving rules is that of restricting the live streaming of football matches only when say the World Cup is going on. Afterwards, some of those restrictions are most likely not needed anymore.

2) Scenario B - Upstream: Scenario B discusses a manufacturing workflow which is hierarchically decomposed into sub-workflows. Consider the $ActivityX$ as a manufacturing process which is decomposed into $A1$ to $A3$ sub-workflows, which are again decomposed into the sub-workflows $A10$ to $A18$ as shown in Fig. 2. Engineers are assigned to monitor the progress of specific (sub-)workflows, i.e. if engineer E is responsible for $A18$, then it receives events related to the progress of $A18$. Access control is necessary here as engineers should only receive events related to the (sub-)workflows they are responsible for. For each non-leaf (sub-)workflow AX, as soon as its children sub-workflows terminate, the parts produced by the children sub-workflows are integrated in AX. If a failure occurs, then a notification is sent to all engineers responsible for its children sub-workflows to resolve the problems among them. This inherently requires that an engineer (say E) receives events related to $A18$ be also allowed to receive events related to all its ancestors $A3$ and $ActivityX$. We refer to this as the upstream pattern. Nowadays, outsourcing is a a very relevant topic for industries and thus, lifecycle management has to make sure that outsourcing and exchanging sub-workflows still lead to consistent access control rules.

B. Contributions

The goal of our work is to enforce distributed access control for publish/subscribe systems deployed across multiple organizations. The proposed framework should:

- Detect and resolve conflicts leading to a consistent set of access control rules.
- Handle any modifications (over time) in rule specifications in an automated fashion.

The problem domain is formalized in Section II. We then make the following contributions:

- Conflict detection (Section III) and resolution (Section IV): We give conflict detection algorithms for both publisher-publisher and publisher-subscriber conflicts. Once a conflict has been detected, different strategies are discussed that can be used to resolve them.
- Lifecycle management (Section V): We then consider life cycle management issues and give algorithms to handle any changes in policies in an automated fashion.
- Experimental evaluation (Section VI): We finally show the efficiency of our algorithms with the help of a reference prototype implementation. Test results show that our algorithms are scalable both from performance and storage perspectives.

II. Problem Formulation

Our system consists of the following actors:

- **Publishers and Subscribers**: Let $\mathcal{P}$ and $\mathcal{S}$ denote the set of publishers and subscribers, respectively.
- **Profiles**: Publishers and subscribers are characterized by their profiles, defined as a set of attribute-value pairs. Let $\mathcal{A}_p$ and $\mathcal{A}_s$ denote the attributes' set applicable to publishers and subscribers, respectively. Each publisher (or subscriber) attribute $a \in \mathcal{A}_p$ (or $a \in \mathcal{A}_s$) is either alphabetic or numeric. Each alphabetic attribute a can take one of the pre-defined values $v \in a_{\mathcal{V}}$. The possible values for a numeric attribute a is defined by the range $a_{min} \leq v \leq a_{max}$. The type of an attribute i.e. whether it is alphabetic or numeric can be determined from its value, and we thus use the same notation a to denote both types of attributes. Given this, the profile of a publisher $P \in \mathcal{P}$ can be specified as a subset of attribute-value pairs $av_p = (a_p, v_p)$: $P.profile = \{av_{p_1}, \cdots, av_{p_i}\}$, where $i \leq |\mathcal{A}_p|$. Subscriber profiles can be specified analogously.

- **Event types hierarchy**: Let $\mathcal{E}$ denote the set of possible event types. We consider a hierarchical relationship among the event types, denoted as $tree(E)$. We say that an event of type $e_1 \in \mathcal{E}$ is a subevent (superevent) of $e_2 \in \mathcal{E}$, if $e_1 = e_2$ or e_2 is an ancestor (descendent) of e_1 in $tree(E)$. A $subtree(e)$ for an event $e \in \mathcal{E}$ can be defined analogously, constituting of e and all descendents of e. In real-life scenarios, e.g. scenarios A and B introduced in Section I-A, it is feasible that the hierarchical organization also imposes some constraints on the publishers and subscribers w.r.t. the event types they can publish and subscribe to, respectively. We consider the following hierarchical constraint scopes:
  - **Downward**: A publisher P (subscriber S) can publish (subscribe to) events of type e implies that P (S) can also publish (subscribe to) all descendent event types of e in $tree(E)$, i.e. the downward scope applies to all events in $subtree(e)$.
  - **Upward**: A publisher P (subscriber S) can publish (subscribe to) events of type e implies that P (S) can also publish (subscribe to) all ancestor event types of e in $tree(E)$.
- **Preferences and Restrictions**: For both publishers and subscribers, only events of specific types are applicable to them. Applicability depends on both their own preferences as well as on restrictions imposed by others. For instance, a publisher P_1 may only be able to publish events of type e due to its functional limitations. However, a competing publisher P_2 may aim to be the only publisher of events e leading to restrictions on what P_1 can publish. Similarly, from the subscriber's perspective, subscription costs money and a subscriber S would only subscribe to event types of its interest. Finally, a subscriber may also have preferences w.r.t. the publishers from which to receive events, and vice versa. The above preferences and restrictions can be specified as follows:
  $P.publish(e, fnc_p, fnc_s)$
  $S.subscribe(e, fnc_p)$
  $fnc_p := av_p \mid fnc_p \lor fnc_p \mid fnc_p \land fnc_p \mid \neg fnc_p$
  $fnc_s := av_s \mid fnc_s \lor fnc_s \mid fnc_s \land fnc_s \mid \neg fnc_s$
  A publisher P uses the rule $P.publish(e,\ fnc_p,\ fnc_s)$ to specify that only (i) publishers whose profiles satisfy fnc_p are allowed to publish events of type e and (ii) subscribers having profiles satisfying fnc_s are allowed to receive events e. The $subscribe()$ function definition follows analogously.

A. Conflict Detection and Resolution

We consider a decentralized and autonomous model where each publisher and subscriber is allowed to specify its own preferences and restrictions. Conflicts are clearly possible in such a scenario. Conflicts are possible both among publishers and also between publishers and subscribers. Conflicts among publishers are captured as follows:

Definition 1 (Publisher-Publisher Conflict): For a pair of publishers $P_1 \neq P_2 \in \mathcal{P}$, their rules $P_1.publish(e_1,\ fnc_{p_1},\ fnc_{s_1})$ and $P_2.publish(e_2, fnc_{p_2}, fnc_{s_2})$ conflict if all the following hold:

1) $e_1 = e_2$ or e_1 and e_2 are hierarchically related, i.e. e_1 is an ancestor or descendent of e_2.
2) A publisher P exists such that $P.profile$ satisfies fnc_{p_1} or fnc_{p_2}, but not both.
3) A subscriber S exists such that $S.profile$ satisfies fnc_{s_1} or fnc_{s_2}, but not both.

$\square$

The first condition is necessary to check if there is some overlap among the events affected by the publish rules of P_1 and P_2. If the second condition does not hold, there exists a publisher whose profile does not satisfy say $fnc_{p_1}(fnc_{p_2})$ but is still able to publish events of type e or of types hierarchically related to e, i.e. $P_1(P_2)$'s publish rule is violated. The third condition performs an analogous check w.r.t. subscribers subscribing to events not allowed by P_1 or P_2. Publisher-subscriber conflicts can be defined on the same lines:

Definition 2 (Publisher-Subscriber Conflict): For a publisher $P \in \mathcal{P}$ and subscriber $S \in \mathcal{S}$, rules $P.publish(e_1,$ $fnc_{p_1},\ fnc_{s_1})$ and $S.subscribe(e_2, fnc_{p_2})$ conflict if $e_1 = e_2$ or e_1 and e_2 are hierarchically related. Further, at least one of the following holds:

- $P.profile$ does not satisfy fnc_{p_2}.
- $S.profile$ does not satisfy fnc_{s_1}.

$\square$

Centralized Access Moderator (CAM): The CAM is responsible for detecting and possibly resolving any conflicts, leading to a globally consistent set of preferences and restrictions. The CAM maintains a database of the consistent set of publish and subscribe rules. However it is the individual publishers and subscribers who are responsible for enforcing the rules, the CAM only acts as a moderator. Each publisher and subscriber first sends its publish and subscribe rules respectively to the CAM for conflict detection. If no conflict is detected, then the rule is added to CAM's database. If a conflict is detected, then a distributed protocol is run among the affected publishers/subscribers to resolve the conflict with the CAM acting as orchestrator. The resolution may involve modification of existing rules in the CAM database, or outright rejection of the proposed rule is also possible.

B. Lifecycle Management

An adaptive publish/subscribe system needs to be able to respond to rules becoming invalid in a seamless and automated fashion. We consider the scenario when a specific rule becomes invalid. Given this, the corresponding fnc_p and/or fnc_s becomes invalid and a common access control rule for an event e may need to be changed.

Definition 3 (Lifecycle Management): For each rule

$$P_1.publish(e_1, fnc_{p_{invalid}}, fnc_{s_{invalid}})$$

from publisher $P_1 \in \mathcal{P}$ becoming invalid, the access control functions $fnc_{p_{invalid}}$ and $fnc_{s_{invalid}}$ from publisher P_1 become invalid as well. Lifecycle management then consists of performing the following actions:

1) Determine a rule $P.publish(e_1, fnc_p, fnc_s)$ from the CAM database which was either rejected or modified earlier. An appropriate rule e.g. would be the last rule to be modified or rejected previously.
2) Check if the rule $P.publish(e_1, fnc_p, fnc_s)$ is in conflict w.r.t. Definitions 1 and 2.
   - If the rule is not in conflict, restore it.
   - If the rule is conflicting, there are the two possibilities: either reject it, or modify it in such a manner that it is not conflicting anymore.
3) If the selected rule was rejected, apply again the lifecycle management algorithm with another previously rejected or modified rule. If no rule got accepted, apply conflict resolution as describer earlier.

□

III. Conflict Detection

A. Data Structures

Alphabetical attributes are denoted as:

$$(Attribute, Attribute\text{-}Value)$$

Numeric attributes represent a range of possible integer values $\{min, max\}$, denoted as:

$$(Attribute, min \mid max, Attribute\text{-}Value)$$

e.g. $(Age, max, 18)$. Attributes are used to specify the profile of both publishers and subscribers. As a number of attributes are needed to describe a profile, we store profiles as list of attributes. By using a list of attributes, the AND relationships between attributes in the list holds implicitly.

$Attr\_list =$
$\{(ATTR_1, ATTR\_V_1), ...(ATTR_N, ATTR\_V_N)\}$
implies that a subscriber has to fulfill all attributes $ATTR_1$, $\cdots ATTR_N$. Rules are stored in the CAM database as:

$$rule(E, P\_list, Attr\_list)$$

where P_list is a list of publishers $[P_1, ...P_n]$ publishing events of type E, and $Attr\_list$ is the list of attributes specifying the subscriber's profile for subscribing to E.

B. Publisher-Publisher Conflicts

Publisher-publisher conflicts occur when two or more publishers would like to enforce separate access control restrictions w.r.t. hierarchically related events. The algorithm given in this section is based on the upwards approach. This implies that a conflict occurs if the $Candidate\_attr\_list$ (abbreviated as $Cattr\_list$) for event E_pub from a publisher P is in conflict with at least one of the $Attr\_lists$ for event E whereby E is equal to E_pub, E_pub_{up}, or E_pub_{down}. The $Cattr\_list$ is in conflict w.r.t. an $Attr\_list$, if one or both lists contain at least one conflicting attribute. An attribute $ATTR$ is conflicting in $List_1$ and $List_2$ if

- attributes $ATTR_1$ in $List_1$ and $ATTR_2$ in $List_2$ differ in their attribute values $ATTR\_VAL$.
- attribute $ATTR$ is a member of $List_1$, but not of $List_2$.

Algorithm 1 detects conflicts by checking for all attributes $ATTRs$ from the $Cattr\_list$ of event E_pub, if the $Cattr\_list$ satisfies all attributes of an existing $rule(E, P\_list, Attr\_list)$ in the CAM database:

1) **Conflicts at the same level** ($E = E\_pub$): The $Cattr\_list$ has to match all attributes and attribute-values from the $Attr\_list$ of event E_pub.
2) **Conflict upwards** ($E = E\_pub_{up}$): The $Cattr\_list$ has to satisfy all attributes and attribute-values from the $Attr\_lists$ of each 'upwards-event' E_pub_{up} in the path from E to the root event in the event types tree. To satisfy an attribute means that all $ATTRs$ and their $ATTR\_VALs$ from the $Attr\_lists$ of E_pub_{up} have to be members of $Cattr\_list$ of event E_pub as well.
3) **Conflict downwards** ($E = E\_pub_{down}$): The $Cattr\_list$ has to be constructed in such a manner that all $Attr\_lists$ of each 'downwards-event' E_pub_{down} in the subtree rooted at E_{pub} in the event types tree satisfies all attributes $ATTRs$ and attribute-values $ATTR\_VALs$ from the $Cattr\_list$ of the event E_pub. To satisfy an attribute means that all $ATTRs$ and their $ATTR\_VALs$ from the $Attr\_list$ of E_pub_{down} have to be members of $Cattr\_list$ of event E_pub as well.

Algorithm 1 $Conflict\_detection()$

```
{/* Conflict at the same level */}
for all $rule(E, P\_list, Attr\_list)$ with $E = E\_pub$ do
    $conflicting\_attributes(Attr\_list, Cattr\_list)$
end for
{/* Conflict upwards */}
for all $rule(E, P\_list, Attr\_list)$ with $E = E\_pub_{up}$ do
    $conflicting\_attributes(Attr\_list, Cattr\_list)$
end for
{/* Conflict downwards */}
for all $rule(E, P\_list, Attr\_list)$ with $E = E\_pub_{down}$ do
    $conflicting\_attributes(Cattr\_list, Attr\_list)$
end for
```

Example 1 (Publisher-Publisher Conflict): Let us assume that two publishers P_1 and P_2 try to add the following five publishing rules in the subsequent order:

1) $rule(A1, P_1, \{(role, engineer)\})$
2) $rule(A1, P_2, \{(company, A)\})$
3) $rule(ActivityX, P_2, \{(company, A)\})$
4) $rule(A12, P_2, \{(role, worker)\})$
5) $rule(A10, P_2, \{(role, engineer), (company, A)\})$

We consider the event types hierarchy as denoted in Fig. 2. This scenario assumes that conflicting access control rules get rejected (even if conflict resolution algorithm is used).

The first publishing rule gets accepted without any conflicts as there are no previously existing rules. When publisher P_2 attempts to enforce the second rule, the CAM detects a 'same level' conflict, because the attribute list specification for subscribers from publisher P_1 contains $(role, engineer)$, but publisher P_2 wants to specify $(company,\ A)$ as attribute list for subscribers. The attributes are conflicting and therefore the rule gets rejected.

On attempting to add event $ActivityX$ from publisher P_2, the CAM detects a downwards conflict, as the publishing rule w.r.t. its downwards-event $A1$ does not satisfy the attribute $(company, A)$. Hence a subscriber would not be able to subscribe to event type $ActivityX$ anymore (rule rejected). When publisher P_2 attempts to enforce the publishing rule for event type $A12$, then the CAM detects an upwards conflict as the publishing rule for $A12$ does not hold the attribute $(role, engineer)$ w.r.t. its upwards-event $A1$. A subscriber for event type $A12$ would no longer be able to subscribe to events of type $A1$ anymore (rule rejected).

While enforcing the last publishing rule for event $A10$ from publisher P_2, the CAM does not detect any conflicts. $A10$ satisfies the attribute $(role, engineer)$ w.r.t. its upwards-event $A1$, and in addition enforces the attribute $(company, A)$ which is not in conflict to any other attribute. □

C. Publisher-Subscriber Conflicts

A publisher-subscriber conflict occurs if a publisher denies a subscriber from receiving its events or none of the publisher profiles match with the preferred publisher profile as preferred by the subscriber. Publisher-subscriber conflicts are detected using the $allow\_subscription()$ algorithm. The algorithm $allow\_subscription(E, P, S)$ checks whether a subscriber S is allowed to subscribe to event E of publisher P based on their profile attributes, or vice versa. The subscriber sends its request to the CAM, which then invokes the function $allow\_subscription()$. The algorithms not given here due to lack of space can be found in the full version: https://sites.google.com/site/debmalyabiswas/research/ConDec_HASE_FV.pdf.

IV. Conflict Resolution

Conflict resolution in an automated fashion is a non-trivial task. In most scenarios there is a need for manual input from an administrator to achieve the desired behavior of the system. Nevertheless, we try to describe a fully automated solution, which may be feasible in certain scenarios.

A. Publisher-Publisher Conflicts Resolution

Given a conflict, the conflicting rules (including the candidate rule) are analyzed to determine an acceptable rule, for instance by adding or removing conflicting attributes. We consider three different resolution strategies:

- *Add attributes*: $fnc_{resolve} = fnc_{common} \wedge fnc_{conflict_1} \wedge fnc_{conflict_2} \wedge ... \wedge fnc_{conflict_n}$, where fnc_{common} is the non-conflicting part and $fnc_{conflict_n}$ refers to the conflicting parts of different publishers. This resolution strategy adds restrictions to the commonly agreed on access control rule. This solution clearly leads to the candidate rule becoming more restrictive.
- *Delete attributes*: $fnc_{resolve} = fnc_{common}$. This approach removes the conflicting attributes when a conflict occurs. Note that this strategy leads to the candidate rule becoming more relaxed than initially intended.
- *Distributed resolution*: If there is a conflicting fnc_{p_i}, then perform a poll among all P_i for e_k and $fnc_{resolve} = fnc_{common} \wedge fnc_{poll}$. A possible strategy here is to add/delete attributes as decided by the majority of involved parties in the poll.

1) Add Attributes: The 'add attributes' resolution strategy is given in Algorithm 2. If a downwards conflict is detected, we have to exchange the whole subtree of event type E which is implemented in a recursive fashion in Algorithm 3. If there is any conflict in the subtree w.r.t. the new attribute, resolution via 'add attributes' is not feasible and the candidate rule will again get rejected.

Algorithm 2 $add\_attributes()$

```
for all $Conflicting\_attr$ do
  if $upwards\_conflict == TRUE$ then
    $add\_conflicting\_attribute()$
    if $conflict\_downwards == TRUE$ then
      $exchange\_attributes(E, conflicting\_attr)$
    end if
  end if
end for
```

Algorithm 3 $exchange\_attributes()$

```
if $conflict\_at\_the\_same\_level == TRUE$ then
  if $conflict\_solvable == TRUE$ then
    for all $E$ and $E_{down}$ do
      $exchange\_attributes(E_{down}, conflicting\_attr)$
    end for
    $add\_conflicting\_attribute()$
    return TRUE
  else
    return FAIL
  end if
end if
```

Example 2 (Conflict resolution - Add attributes): We consider a CAM database consisting of the following rules:

```
rule(ActivityX,P1,{(company,A)})
rule(A1,P2,(role,engineer),(company,A))
rule(A12,P4,{(role,engineer),(company,A)})
```

Publisher $P2$ then requests the rule

$$requested\_rule(A1, P3, (machine, M))$$

This leads to an upwards conflict w.r.t. $(company, A)$ of $ActivityX$, and a conflict at the same level w.r.t. $(role, engineer)$ and $(machine, M)$ of event $A1$. In addition, it implies a downwards conflict w.r.t. $(machine, M)$ of event $A12$. The conflict can be resolved by adding $(machine, M)$ to the rule for event types $A1$ and $A12$.

□

Algorithm 4 $delete\_attributes()$

```
if $conflict\_detection == FALSE$ then
    $updated\_rule = delete\_attribute\_from\_rule()$
    for all $E_{down}$ do
        $delete\_attribute(E_{down}, (Attr, Attr\_Val))$
    end for
    return $updated\_rule$
end if
```

2) Delete Attributes: The 'delete attributes' strategy is given in Algorithm 4.

We suffice to say that the third strategy can be implemented by extending the above with a distributed agreement protocol.

B. Publisher-Subscriber Conflicts Resolution

Publisher-Subscriber conflicts imply that there is a mismatch between publisher and subscriber preferences. This essentially is a matchmaking problem and approaches to resolve such conflicts are out of the scope of this work.

V. Lifecycle Management

In this section, we present our lifecycle management algorithms. The intuition behind our lifecycle management framework is to maintain a history of access control rule updates including rejections. When publishers tag an access control rule as invalid, e.g. when a rule has outlived its purpose, the CAM can either rollback to a previously modified accepted database state or incorporate previously rejected rules which now may become valid.

A. Maintain Access Control Rules History

For recording the history of our publish/subscribe system rule updates we need to store two types of data. As we are interested in the 'most recent' (other heuristics can be accommodated analogously) rejected/modified access control rule for life cycle management, we maintain our log in the form of a stack. We record the following history elements:

- **Rejected Access Control Rules**: Every time a publisher tries to enforce its referenced access control policy w.r.t. event E, we record it as follows:

$$log(rejected\_rule(E, P, Requested\_attr\_list))$$

- **Modified Accepted Access Control Rules**: When an inquiry for adding an access control rule was successful, we store it as follows:

$$log(modified\_rule(E, P\_list, Attr\_list))$$

B. Remove Publishing Rule

We next give the algorithm to handle removal of a rule from the CAM database, as a result of the rule becoming invalid or the publisher/subscriber who had originally specified the rule going offline. As mentioned earlier, we have two options w.r.t. adapting the existing CAM database when a rule is removed:

- **Rollback to a rule from the log**: The preferred option is to take rules from the log and rollback to either a previously accepted or rejected rule which can become valid now. There are clearly many options w.r.t. which access control rule to consider for reactivation, e.g. the the most *recently rejected* rule, We need to ensure that any such rule selected for reactivation is not in conflict with the current CAM database (after removal of the rule in question). So we have to again check for conflicts. If no conflicts exist, the candidate rule can be added to the CAM database. Else we try with the next most suitable rule, and so no. If all candidate rules are conflicting, we proceed with the alternate strategy 'modify current rules'.
- **Modify current rules**: Here we take the current rule and modify it in such a manner that it is no longer conflicting with the current rule set. To achieve this, we first delete all attributes from the lists $Attr\_list\_to\_delete$, and then try to add previously rejected attribute requests which may have a chance of becoming valid now. **Remove outdated attributes** (Algorithm 4): If a publisher invalidates his rule, his restricted attributes have to be deleted from the common rule. But this is only allowed if there is no conflict w.r.t. restricted attributes from other publishers specifying access control rules for the same or hierarchically related event types. **Add previously rejected attributes** (Algorithm 2): When an attribute becomes invalid, previously rejected attributes may become valid now. Note that adding previously rejected attributes is not required for consistency, but is a desirable feature in that more requests can be accommodated.

Algorithm 5 $removing\_publishing\_rule()$

```
{/* Rollback to a rule from the log */}
$Candidate\_set = rules\_for\_reactivation\_from\_log()$
for all $Candidates \in Candiate\_set$ do
    if $conflict\_detection() == FALSE$ then
        $delete\_current\_rule()$
        $add\_candidate\_to\_cam()$
        $clear\_history()$
        return $rule$
    end if
end for
{/* Delete outdated attributes */}
for all $(Attr, Attr\_Val) \in Attr\_list\_to\_remove$ do
    $delete\_attributes(E, (Attr, Attr\_Val))$
end for
{/* Add previously rejected attributes */}
for all $(Attr, Attr\_Val) \in rejected\_attr\_list\_log()$ do
    $add\_attributes(E, (Attr, Attr\_Val))$
end for
$update\_accepted\_access\_control\_rule\_log()$
return $rule$
```

Example 3 (Rollback to a rule from the log): We consider the event types hierarchy in Fig. 2, and the following log:

```
rule(A1,[P1,P2],{(company,A),(role,engineer)})
log(modified_rule(A1,P2,{(company,A)}))
log(modified_rule(A1,P1,{(role,engineer)}))
log(rejected_rule(A1,P2,{(role,worker)}))
rule(A1,[P1],{(role,engineer)})
rule(A12,[P3],{(role,engineer),(company,A)})
```

Now let us assume that publisher $P2$ invalidates his rule w.r.t. $A1$. The CAM then performs a rollback to

$$modified\_rule(A1, P1, \{(role, engineer)\})$$

which is not in conflict with publisher $P3$'s rule w.r.t. $A12$.

If publisher $P1$'s rule further becomes invalid, the CAM performs a rollback to:

$$modified\_rule(A1, P2, \{(company, A)\})$$

which is again not in conflict with the related rule of publisher $P3$ w.r.t. $A12$. Note that it is not possible to perform a rollback to the rejected rule specifying the attribute $(role, worker)$ because this would lead to a downwards conflict w.r.t. $A12$. $\square$

VI. Experimental Evaluation

For our reference implementation, we chose Prolog as the programming language. Prolog supports relations which are represented by facts and rules. Facts and rules are stored in an internal knowledge database. Users can run queries over these relations and the Prolog interpreter evaluates them to show results that satisfy the given goal.

In our system, the access control policies that get accepted are maintained as facts in the Prolog database. The attribute based profiles of publishers and subscribers are also modeled as facts in the Prolog database. An access control decision is performed as a query to the Prolog interpreter.

We used SWI-Prolog version 5.10.1 as our Prolog interpreter which is distributed under LGPL. SWI-Prolog supports an additional package for unit testing that we used for our test cases. The entry points for every unit test are defined by rules using the head $test(Test\_name)$. Note that a test which is supposed to fail is denoted with \+ which is the Prolog symbol for 'NOT'. All tests were performed on a computer with an Intel Pentium Mobile 2 GHz and 1024 MB RAM running Ubuntu 10.04 as operating system.

The tests were performed in a specific order as some cases require specific events to have already occurred in the past. We give a sample scenario below illustrating the tool output. We first add a rule specifying the attribute $(role, manager)$ for event $a1$ from publisher $p1$ to the empty CAM database.

```
test(add_publishing_rule1):-
add_publishing_rule(a1,p1,[(role,manager)]).
```

Output:

```
Rule successfully added.
```

We then test for conflicts at the same level.

```
test(add_publishing_rule2):-
\+ add_publishing_rule(a1,p2,[(company,a)]).
```

Output:

```
Failed to add rule: Conflict at the same level!
Missing attribute(s): (role,manager)
```

With conflict resolution and lifecycle management enabled:

Output:

```
Failed to add rule: Conflict at the same level!
Missing attribute(s): (role,manager)
Conflict resolution started.
Rule successfully resolved:
rule(a1,[p1,p2],[(company,a),(role,manager)])
and
rule(a1,p1,[(role,manager)]) added to history_log.
```

Now we consider a (future) change action where an existing rule is declared as invalid.

```
test(delete_rule1):-
delete_rule(a1,p2,[(company,a)]).
```

Output:

```
Rollback to rule: rule(a1,p1,[(role,manager)])
Delete from history_log:
rule(a1,p1,[(role,manager)])
Add to history_log:
rule(a1,[p1,p2],[(company,a),(role,manager)])
```

We now give sample run times for our algorithms to illustrate their scalability. We have developed an automated test case generator which outputs scenarios based on 3 input parameters: (i) number of rules r, (ii) number of publishers $p = |\mathcal{P}|$, and (iii) number of event types (nodes in the event tree) $n = |\mathcal{E}|$. The generated event tree is balanced, such that $n = 2^l - 1$ where l is the number of levels in the event tree.

Fig. 3 shows the execution time as the number of rules increases for $n = 105$ and $p = 100$. The increase in execution time is (approx.) linear and the time $7830ms$ to evaluate 10000 rules clearly shows the scalability of our algorithms. Fig. 4 shows the execution time as the number of publishers increases for $n = 105$ and $r = 1000$. It can be seen that increasing the number of publishers does not really affect the execution time. This is because publishers act as peers in the current setup with equal capabilities. The number of publishers p would become relevant if we considered different publishers having different CPU/communication overheads. Fig. 5 shows the execution time as the size of the event tree increases for $p = 105$ and $r = 100$. The increase in execution time is logarithmic, which should be acceptable in practice as we do not expect the number of event types to be too large even for large scale real-life scenarios.

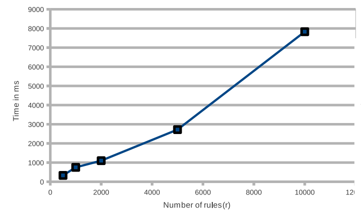

Fig. 3. Ext. time vs. Rules

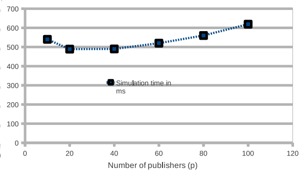

Fig. 4. Ext. time vs. Pubs

VII. Related Work

In contrast to current research in access control models for publish/subscribe systems [2], [3], [4], [5], we provided

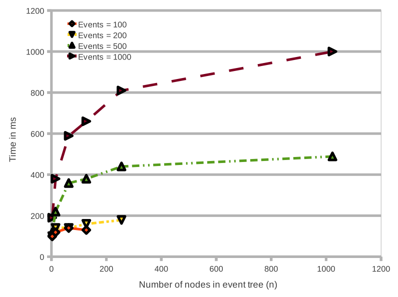

Fig. 5. Ext. time vs. Size of the event tree

a comprehensive framework to support conflict detection, resolution and lifecycle management.

We adapted an ABAC scheme [6] for specifying access control rules in publish/subscribe systems. The attribute based authorization framework proposed in [7] offers dynamic conflict resolution. Depending on the attribute in question, the framework chooses a feasible policy combining algorithm. For instance, if the attribute 'emergency' in a hospital environment applies, users will gain additional access control rights they normally do not get. This is a practical approach, but problems will arise in more complex scenarios when more than one attribute applies. A related approach here is also the work on Attribute Based Encryption (ABE) [8]. ABE is a cryptographic scheme for decentralized access control where attributes/profiles of parties allowed to decrypt a ciphertext, are embedded in the ciphertext itself. Given this, while the part which deals with determining if a party can decrypt a ciphertext is analogous to our conflict detection aspect, resolution and lifecycle management are clearly missing here.

We made use of hierarchical relationships prevalent among event types to optimize our algorithms. [9] gives a graph-based conceptual model to capture such hierarchical relationships in a multi-provider scenario. Policy decomposition for collaborative access control is considered in [10]. They propose to decompose a global access control policy rule into local rules and then distribute it to each collaborating party. Here we take the reverse approach of allowing each party to first come up with their own preferred local rules, and then compose them to a globally consistent non-conflicting set of rules.

Conflict detection using description logic based methods is proposed in [11], [12], [13]. They first show how to model access control policies in description logic, followed by algorithms for policy comparison, verification and querying. However none of them deal with conflict resolution or lifecycle management. For designing conflict resolution algorithms, one of our strategies is influenced by the notion of computing rule similarities [14] between XACML based access control policies. Conflict resolution is also discussed in [15]. They propose to either resolve conflicts by applying only permit-overrides (deny-overrides) or by using an algorithm based on priority for certain access control rules, however the actual implementation details to achieve this in practice are missing.

VIII. Conclusions and Future Work

We considered the problem of access control in large-scale publish subscribe systems, deployed across multiple organizations. In such a scenario, the publishers and subscribers prefer to autonomously specify and enforce their access control policies leading to conflicts. We developed conflict detection and resolution algorithms for both publisher-publisher and publisher-subscriber conflicts ensuring a mutually consistent set of non-conflicting rules. We further showed how the hierarchical relationships among event types can not only be accommodated, but also used to increase the efficiency of our access control algorithms. We next provided algorithms for automated lifecycle management that allow rollback to a previously consistent set of rules. We showed the scalability of our algorithms by evaluation on our reference implementation.

In future, we would like to be able to accommodate changes in the hierarchical event model itself. It has to be seen how deleting/adding event types affects the conflict detection and lifecycle management algorithms.

References

[1] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The Many faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[2] C. Wang, A. Carzaniga, D. Evans, and A. Wolf, "Security Issues and Requirements for Internet-scale Publish-subscribe Systems," in *Hawaii International Conference on System Sciences*, 2002, pp. 3940–3947.

[3] A. Belokosztolszki, D. Eyers, P. Pietzuch, J. Bacon, and K. Moody, "Role-based Access Control for Publish/Subscribe Middleware Architectures," in *International Workshop on Distributed Event-based Systems*, 2003, pp. 1–8.

[4] Y. Zhao and D. Sturman, "Dynamic Access Control in a Content-based Publish/Subscribe System with Delivery Guarantees," in *International Conference on Distributed Computing Systems*, 2006.

[5] J. Singh, L. Vargas, J. Bacon, and K. Moody, "Policy-based Information Sharing in Publish/Subscribe Middleware," in *Workshop on Policies for Distributed Systems and Networks*, 2008, pp. 137–144.

[6] E. Yuan and J. Tong, "Attributed based Access Control (ABAC) for Web Services," in *IEEE International Conference on Web Services*, 2005, pp. 561–569.

[7] A. Mohan and D. Blough, "An Attribute-based Authorization Policy Framework with Dynamic Conflict resolution," in *Symposium on Identity and Trust on the Internet*, 2010, pp. 37–50.

[8] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-Policy Attribute-Based Encryption," in *IEEE Symposium on Security and Privacy*, 2007, pp. 321–334.

[9] D. Biswas and K. Vidyasankar, "Formalizing Visibility Characteristics in Hierarchical Systems," *Data and Knowledge Engineering*, vol. 68, no. 8, pp. 748–774, 2009.

[10] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo, "Policy Decomposition for Collaborative Access Control," in *ACM Symposium on Access Control Models and Technologies*, 2008, pp. 103–112.

[11] V. Kolovski, J. Hendler, and B. Parsia, "Analyzing Web Access Control Policies," in *International Conference on World Wide Web*, 2007, pp. 677–686.

[12] V. Kolovski, "Formalizing XACML Using Defeasible Description Logics," University of Maryland, USA, Tech. Rep. TR-233-11, 2006.

[13] D. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and Reasoning about Dynamic Access-control Policies," *Automated Reasoning*, pp. 632–646, 2006.

[14] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino, "XACML Policy Integration Algorithms," *ACM Transactions on Information and System Security*, vol. 11, no. 1, pp. 1–29, 2008.

[15] W. Yu and E. Nayak, "An Algorithmic Approach to Authorization Rules Conflict Resolution in Software Security," in *IEEE International Computer Software and Applications Conference*, 2008, pp. 32–35.