

Automatic Support for Policy Creation

# **Conflict Detection and Resolution of XACML Policies**

Florian Huonder, fhuonder@herasaf.org,  
Master Thesis, University of Applied Sciences Rapperswil

July 2010

Supervisor: Prof. Dr. Josef Joller

Expert: Dr. Günter Karjoth (IBM, Zurich Research Laboratory)

## Abstract

Enterprises and other organizations are making more and more use of XACML-based access control solutions. An administrator who is responsible for the creation and maintenance of the access control policies faces a lot of problems that are insufficiently solved up to now. One of these problems is the requirement to deploy a set of policies that is free of any conflicts.

With this thesis we propose algorithms for *Conflict Detection* and *Conflict Resolution* that enable an administrator to easily detect conflicts among the created policies and resolve those conflicts in a reasonable way.

## Acknowledgements

First of all I want to thank Prof. Dr. Josef Joller for his great support during this thesis. We had a lot of fruitful discussions and his comments on early reviews brought good improvements.

Second I want to thank the whole HERAS<sup>AF</sup> core team for their support. Especially I want to thank Stefan Oberholzer for his input regarding the algorithms and also for cross-checking their correctness. I also want to thank Wolfgang Giersche for his input during the whole thesis.

Third I want to thank Günter Karjoth from IBM for his great contribution during the milestone meetings.

Last but not least I want to thank my family and friends for their support and patience during the long time of this master thesis.

# Contents

<b>I. Inception</b>	<b>6</b>
I.1. Introduction . . . . .	7
I.2. Related Work . . . . .	9
I.3. XACML in a Nutshell . . . . .	10
I.3.1. XACML Policy Model . . . . .	11
I.4. Notation . . . . .	13
I.5. Basic Concepts . . . . .	16
I.6. Algebraic Approach . . . . .	17
I.6.1. Fine-Grained Integration Algebra . . . . .	17
I.6.1.1. Basic Definitions . . . . .	17
I.6.1.2. XACML Combining Algorithms . . . . .	20
I.7. Geometric Approach . . . . .	21
I.7.1. XACML Policy as N-Dimensional Rectangle . . . . .	21
I.7.1.1. Basic Definitions . . . . .	21
I.7.1.2. Example . . . . .	21
I.8. Algebraic Approach compared to the Geometric Approach . . . . .	23
<b>II. Conflict Detection</b>	<b>24</b>
II.1. Definition of Conflict . . . . .	25
II.2. Our Approach . . . . .	27
II.3. Mapping XACML Target to the N-Dimensional Space . . . . .	28
II.3.1. Attribute Types mapped to Integer . . . . .	28
II.3.2. Attribute Types mapped to Double . . . . .	29
II.3.3. Attribute Types mapped to String . . . . .	29
II.3.4. Attribute Types mapped to Finite Automata . . . . .	30
II.4. Conflict Detection Algorithm . . . . .	35
II.4.1. Determining Start- and End-Point of the range a policy covers . . . . .	35
II.4.2. Determine all pairwise intersections . . . . .	36
II.4.2.1. Plane Sweep Algorithm . . . . .	37
II.4.2.2. AABB - AABB Intersection Algorithm . . . . .	38
II.4.2.3. Regex Intersection Algorithm . . . . .	41
II.4.3. Pruning all non-intersecting policies . . . . .	43
II.4.4. Report all pairwise conflicts . . . . .	44
II.4.5. Implementation Hints - Data Structures . . . . .	44
II.4.5.1. Intersection-Matrix . . . . .	44
II.4.5.2. Global Policy List . . . . .	46
II.4.6. Runtime and Application Analysis . . . . .	46
II.4.6.1. Runtime Analysis . . . . .	47
II.4.6.2. Application Analysis . . . . .	49

II.4.7. Runtime Optimization . . . . .	49
II.4.7.1. Clustering . . . . .	49
II.4.7.2. Attribute Ordering . . . . .	50
<b>III. Conflict Resolution</b>	<b>51</b>
III.1. Directed Acyclic Graph of Resolutions . . . . .	52
III.1.1. Cycle Detection . . . . .	54
III.2. Resolution Strategies . . . . .	55
III.2.1. Cutting Planes Algorithm . . . . .	55
III.2.1.1. Algorithm . . . . .	56
III.2.1.2. Examples . . . . .	58
III.2.2. Precedence Stringing . . . . .	66
III.2.2.1. Example . . . . .	66
III.2.3. Analysis . . . . .	68
III.2.3.1. Cutting Planes . . . . .	68
III.2.3.2. Precedence Stringing . . . . .	68
III.2.3.3. Comparison . . . . .	68
III.3. Default Decision . . . . .	69
<b>IV. Conclusion</b>	<b>70</b>
IV.1. Achievements . . . . .	71
IV.2. Further Work . . . . .	72
<b>A. Examples</b>	<b>73</b>
A.1. FIA Examples . . . . .	74
A.1.1. Addition . . . . .	76
A.1.2. Intersection . . . . .	77
A.1.3. Negation . . . . .	77
A.1.4. Domain Projection . . . . .	78
A.1.5. Effect Projection . . . . .	78
A.1.6. Subtraction . . . . .	79
A.1.7. Precedence . . . . .	80

# I. Inception

## 1.1. Introduction

The *eXtensible Access Control Markup Language 2.0* (XACML 2.0) [2] is an OASIS<sup>1</sup> approved standard for access control. The XACML 2.0 specification describes the syntax of policies, requests and responses. Further it describes how requests must be evaluated against the policies to get a proper response. The XACML 2.0 specification also provides a set of combining algorithms used to resolve policy conflicts. Conflicting policies are policies that are applicable to the same request but result in different responses. Such algorithms are e.g. *permit-overrides-algorithm*, *deny-overrides-algorithm*, *first-applicable-algorithm* and so on. These algorithms resolve conflicts but do not care about the correctness of the resolution. XACML does anyway not have the ability to make any semantical decision based on the given attributes. It is up to the responsible personnel (we call him/her *administrator* later on) to design the policies in such a way that the semantic is respected within the conflict resolution.

This thesis introduces algorithms for *Conflict Detection* and *Conflict Resolution* that enable an administrator to detect conflicts and to resolve these conflicts in a sensible way. These algorithms are intended to support an administrator during the policy creation phase. The algorithms take as input a given set of XACML policies that are not combined under any combining algorithm. The algorithms follow a bottom up approach of creating a policy tree that is conflict free.

The two *Conflict Detection Algorithms*, *Plane Sweep Algorithm* and *AABB - AABB Intersection Algorithm* are both geometric algorithms that make use of the fact that XACML policy *Targets* can be expressed as  $n$ -dimensional rectangles. Both are well established algorithms for finding intersections (i.e. conflicts) among geometrical figures (here rectangles). These algorithms are described in Chapter II. Additionally the *Regex Intersection Algorithm* handle regular expressions. We propose a method of finding intersections between regular expressions. This enables the algorithms to detect all conflicts that might occur between XACML policies. Further we also propose some basic ideas about data structures that improve the runtime of our algorithms.

The two *Conflict Resolution Algorithms*, *Cutting Planes Algorithm* and *Precedence Stringing Algorithm*, both follow a basically different approach of resolving conflicts. Both rely on a *directed acyclic graph* that represents the required conflict resolutions, where the nodes are the policies and the directed edge indicate the resolutions.

This thesis is structured into three chapters. Chapter I contains related work, notation, basic concepts and a description of how XACML policies can be expressed as  $n$ -dimensional rectangles. Chapter II describes the two *Conflict Detection Algorithms* in detail with some examples. Chapter III covers the two *Conflict Resolution Algorithms* in detail with some examples.

There exist two special cases, *policy references* and *multi-valued attributes*.

*Policy references* are references to policies that are deployed on a remote system and are therefore not available. Therefore the *Conflict Detection Algorithms* are not able to include referenced policies into the detection.

*Multi-valued attributes* are attributes that may hold multiple attribute values at the same time. Consider a policy that is applicable for the *role = administrator* and another policy applicable for the *role = user*. A request now could hold an attribute that has

---

<sup>1</sup><http://www.oasis-open.org>

$role = \{administrator, user\}$ . The problem is that the two policies are not considered to conflict but would be applicable at the same time to the given request. This means they may be conflicting though.

These two special cases are excluded in this thesis. We assume that every policy in the system is available and that no multi-valued attributes exist. We suggest some ideas how to face those problems in Section IV.2.

## I.2. Related Work

Until now there was no work done on algorithms that support an administrator during the creation of access control policies, regarding the conflict detection and resolution. There was a lot of work done regarding the composition of two policy sets (e.g. in case of a merger), as discussed by Rao et al. [17]; it is discussed how policies can be combined if two or more companies start sharing common resources. In Section I.6.1 we take a closer look at the *Fine-Grained Integration Algebra*.

A similar work was done by Bonatti et al. [1] that addresses the problem of combining authorization specifications that may be independently stated, possibly in different languages and according to different policies. It is about the same topic but goes one step further than discussed by Rao et al. [17] due to the fact that policies in arbitrary languages are respected.

### I.3. XACML in a Nutshell

XACML [2] is an OASIS<sup>2</sup> standard for access control. It describes both a policy language and an access control decision request/response language (both written in XML). XACML defines various extension points for defining new functions, data types, combining logic and so on.

An access control request is sent to a decision point that evaluates the request against the deployed policies and returns a response. The response includes the answer on the request with one of the four values *Permit*, *Deny*, *Indeterminate* (an error occurred) or *Not Applicable* (no decision can be taken).

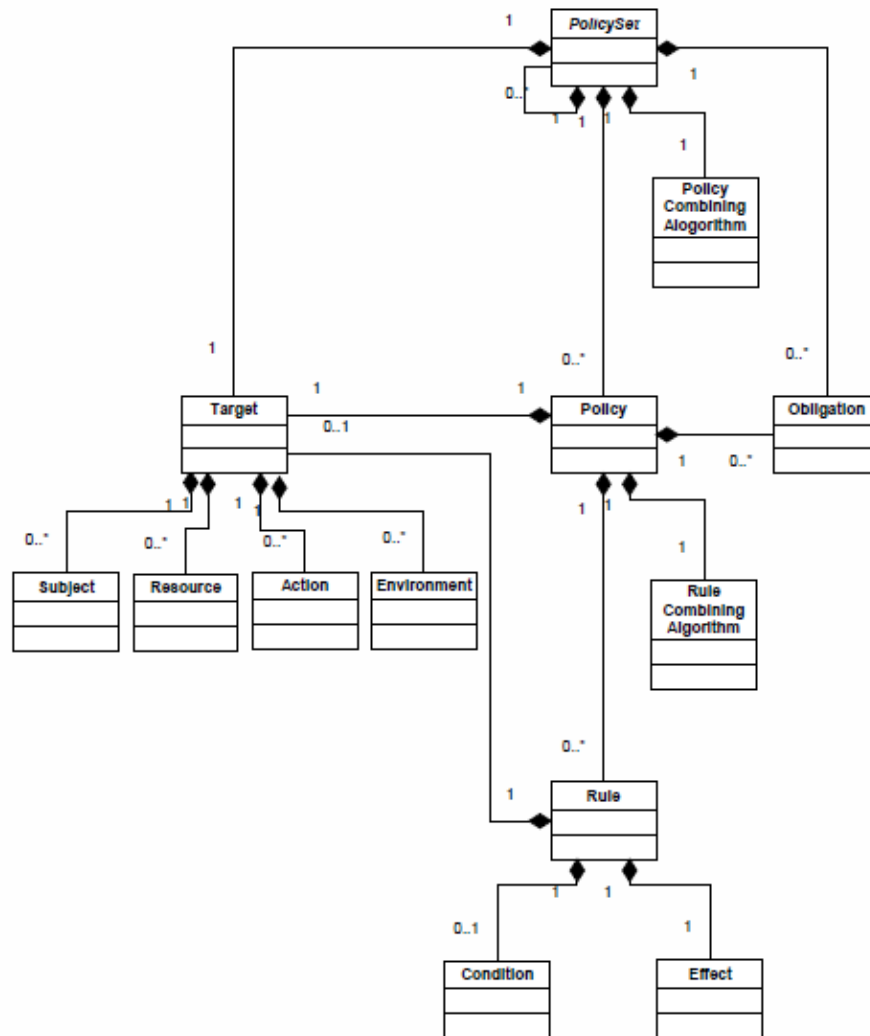


Figure I.1.: XACML Policy Model [2]

The requests are generated by a *Policy Enforcement Point* (PEP) and sent to the decision point, the *Policy Decision Point* (PDP). The PEP is, as its name already implies,

<sup>2</sup><http://www.oasis-open.org>

responsible of enforcing the PDP's decision. The advantages of XACML are<sup>3</sup>:

- **It's standard.** By using a standard language, you're using something that has been reviewed by a large community of experts and users, you don't need to roll your own system each time, and you don't need to think about all the tricky issues involved in designing a new language. Plus, as XACML becomes more widely deployed, it will be easier to interoperate with other applications using the same standard language.
- **It's generic.** This means that rather than trying to provide access control for a particular environment or a specific kind of resource, it can be used in any environment. One policy can be written which can then be used by many different kinds of applications, and when one common language is used, policy management becomes much easier.
- **It's distributed.** This means that a policy can be written which in turn refers to other policies kept in arbitrary locations. The result is that rather than having to manage a single monolithic policy, different people or groups can manage sub-pieces of policies as appropriate, and XACML knows how to correctly combine the results from these different policies into one decision.
- **It's powerful.** While there are many ways the base language can be extended, many environments will not need to do so. The standard language already supports a wide variety of data types, functions, and rules about combining the results of different policies. In addition to this, there are already standards groups working on extensions and profiles that will hook XACML into other standards like SAML and LDAP, which will increase the number of ways that XACML can be used.

### 1.3.1. XACML Policy Model

Hereafter we introduce the policy language model of XACML. Figure I.1 on page 10 shows an overview of the most important components. These are described hereafter:

- **PolicySet** This element may contain multiple *Policy* elements and *Obligations* and contains a *Target*.
- **Policy** This element may contain multiple *Rule* elements and *Obligations* and contains a *Target*.
- **Rule** This element may contain a *Condition* and a *Target* and contains an *Effect*.
- **Target** This element contains the attributes that a request must match so that the *PolicySet*, *Policy* or *Rule* element that contains this *Target* is applicable.
- **Subject** This element contains the subject's attributes of the *Target*.
- **Resource** This element contains the resource's attributes of the *Target*.
- **Action** This element contains the action's attributes of the *Target*.

---

<sup>3</sup>From <http://www.oasis-open.org>

- **Environment** This element contains the attributes that do not match to *Subject*, *Resource* or *Action* of the *Target*.
- **Condition** This element contains functions that are applied to the attributes to check for applicability of a request. This is more restrictive than the *Target* element.
- **Effect** This element is either *Permit* or *Deny* and states to what a *Rule* evaluates if the *Target* matches and the *Condition* returns true.
- **RuleCombiningAlgorithm** This element states the rule under which applicable *Rules* with different *Effects* are combined.
- **PolicyCombiningAlgorithm** This element states the rule under which applicable *Policies* with different decisions (i.e. *Effects* of their *Rules*) are combined.
- **Obligation** This element contains the *Obligations* that a requestor must fulfill so that the decision can be enforced.

In our terms we use the name *policy* to describe either the *Target* element or one of the elements *PolicySet*, *Policy* or *Rule*. We do not differentiate between them because for the topic of conflict detection and conflict resolution those are interchangeable. The reason is that we are only interested whether the *Targets* overlap, this means whether more than one *Target* is applicable to the same request. Further we only consider *Rule* elements because *Policy* and *PolicySet* elements are only containers that realize the resolution of a conflict.

## I.4. Notation

In this section we introduce the needed notations, see Table I.1 on page 14. Our notation describes how a standard XACML *Target* must be treated to be able to determine conflicts among policies. We use the formalism introduced by Oberholzer [16] to express policy *Targets*. The structure of an XACML *Target* can be seen in Listing I.1.

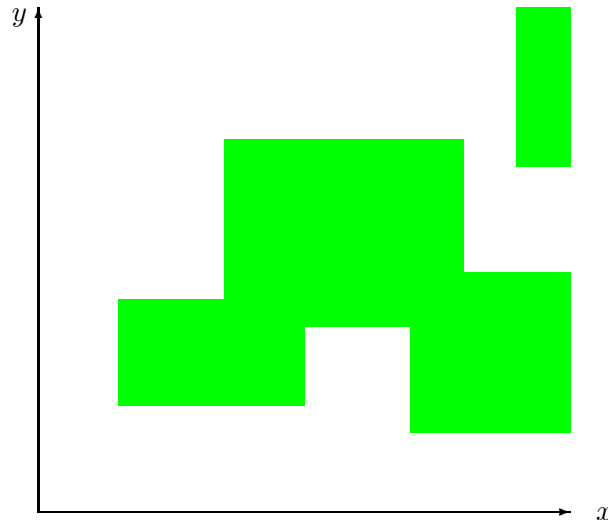


Figure I.2.: XACML Target with *OR* composition.

```

<Target> <!-- Policy Target (PT) -->
  <Subjects> <!-- Target-Group (TG) -->
    <Subject> <!-- Matching-Group (MG) -->
      <SubjectMatch> <!-- Matching-Element (ME) -->
        <AttributeValue> <!-- Matching-Value (MV) -->
        <SubjectAttributeDesignator>
      <Resources>
        <Resource>
          <ResourceMatch>
            <AttributeValue>
            <ResourceAttributeDesignator>
          <Actions>
            <Action>
              <ActionMatch>
                <AttributeValue>
                <ActionAttributeDesignator>
              <Environments>
                <Environment>
                  <EnvironmentMatch>
                    <AttributeValue>
                    <EnvironmentAttributeDesignator>

```

Listing I.1: XACML Target structure

An XACML *Target* can be interpreted as an orthogonal figure. To be able to detect conflicts this figure is required to be a rectangle, see Section II.1 for further details. We use and extend<sup>4</sup> the formalism introduced by Oberholzer [16] to be able to split an XACML *Target*

<sup>4</sup>The extension is required because Oberholzer [16] did not respect the OR semantics of the *attribute values*.

into multiple rectangles, called *Virtual Targets*. The reason to split an XACML *Target* into multiple *Virtual Targets* is that a *Target*, due to the *OR* semantics, may look like Figure I.2 on page 13. With transforming the *Targets* into *Virtual Targets* they are normalized to rectangles and this enables the further handling.

XACML element	Our notation	
<Target>	$PT^i$	where $i$ is the identifier of the parent policy, policy set or rule of this <i>Target</i>
<Subjects>	$TG^{subject,i}$	
<Resources>	$TG^{resource,i}$	
<Actions>	$TG^{action,i}$	where $i$ is the same as above
<Environments>	$TG^{environment,i}$	
<Subject>	$MG_k^{subject,i}$	where $i$ is the same as above
<Resource>	$MG_k^{resource,i}$	and $k$ the identifier of the matching-group within that target-group
<Action>	$MG_k^{action,i}$	
<Environment>	$MG_k^{environment,i}$	
<SubjectMatch>	$ME_{k,m}^{subject,i}$	where $i$ and $k$ are the same as above
<ResourceMatch>	$ME_{k,m}^{resource,i}$	and $m$ the identifier of the matching-element within that matching-group
<ActionMatch>	$ME_{k,m}^{action,i}$	
<EnvironmentMatch>	$ME_{k,m}^{environment,i}$	
<AttributeValue>	$MV_{k,m,v}^{subject,i}$	where $i$ , $k$ and $m$ are the same as above
<AttributeValue>	$MV_{k,m,v}^{resource,i}$	and $v$ the identifier of the matching-value within the given matching-element
<AttributeValue>	$MV_{k,m,v}^{action,i}$	
<AttributeValue>	$MV_{k,m,v}^{environment,i}$	

Table I.1.: Mapping of XACML Targets.

The semantics of an XACML *Target* is

$$PT^i = TG^{subject,i} \wedge TG^{resource,i} \wedge TG^{action,i} \wedge TG^{environment,i}$$

$$TG^{x,i} = MG_0^{x,i} \vee MG_1^{x,i} \vee \dots \vee MG_k^{x,i}$$

$$MG_k^{x,i} = ME_{k,0}^{x,i} \wedge ME_{k,1}^{x,i} \wedge \dots \wedge ME_{k,m}^{x,i}$$

$$ME_{k,v}^{x,i} = MV_{k,v,0}^{x,i} \vee MV_{k,v,1}^{x,i} \vee \dots \vee MV_{k,v,n}^{x,i}$$

The definitions of the elements are in Table I.1 on page 14. Mapping an XACML *Target* to an  $n$ -dimensional space may, because of the *OR* semantics of the *matching-group* ( $MG$ ) and *matching-value* ( $MV$ ), result in an  $n$ -dimensional orthogonal figure that is not a rectangle. To be able to apply the conflict detection algorithms defined in Section II.4 we use the

*Virtual Target* ( $VirtualT_j^i$ ). A *Virtual Target* consists of all possible *and*-combinations of *matching-groups* ( $MG$ ) and *matching-values* ( $MV$ ).

## I.5. Basic Concepts

This section is about the structure of XACML. The most important elements and their relation is described in Section I.3. An XACML policy may be a *Policy* or a *PolicySet* where a *Policy* contains a *Rule*. Each of these elements has a *Target* that states the applicability to a certain set of requests.

In this thesis we use the term policy interchangeably to name one of the above elements. When we talk of conflicting policies we mean conflicting *Targets*. Further we assume that the unit under inspection (for conflict detection) is only the *Target* of a *Rule*. We do only consider single access control statements in form of a single *Rule* contained in a single *Policy*<sup>5</sup>. The *Condition* of the *Rule* is ignored due to the fact that the *Condition* is always a harder restriction than the *Target* is. This means if a conflict is detected on *Target* level it is possible that the whole *Rule* (including the *Condition*) would not conflict. Those cases are neglected.

The first step, prior to the conflict detection, is to split the *Target* of the *Rule* into *Virtual Targets*, see Section I.4. Then the *Rule* is copied and each copy gets one of the *Virtual Targets* as new *Target*. Afterwards each *Rule* has a *Target* that can be mapped to a  $n$ -dimensional rectangle, in our geometric approach.

If a *Rule* has to be split into multiple *Virtual Targets* it makes sense to move these *Virtual Targets* into *PolicySets* and these *PolicySets* then reference the *Policy* that contains the *Rule*. This has the advantage that the *Rule* (also the *Condition*) and the *Obligations* must not be duplicated.

---

<sup>5</sup>The *Policy* is needed because only a *Policy* can have *Obligations*

## I.6. Algebraic Approach

### I.6.1. Fine-Grained Integration Algebra

This section is an overview of the Fine-Grained Integration Algebra [17]. This section is about the basics of the Fine-Grained Integration Algebra (FIA) used to describe XACML policies. The Fine-Grained Integration Algebra is an algebra that can be used to handle the merging of two or more policies. This section is only a summary. Detailed XACML examples can be found in Appendix A.1.

#### I.6.1.1. Basic Definitions

- All attributes are considered as *Target* attributes
- $Y$  denotes the set of attributes that evaluate to *Permit*
- $N$  denotes the set of attributes that evaluate to *Deny*
- $NA$  denotes the set of attributes that evaluate to *Not Applicable*
- $\Sigma$  denotes the vocabulary containing all possible attribute names and their corresponding domain
- Let  $a_i$  be an attribute name and  $v_i \in \text{dom}(a_i)$  an attribute value<sup>6</sup>.
- A request is defined as a set  $r$  of attributes.  $R_\Sigma$  is the set of all possible requests.
- A policy can be considered as a function mapping each request to a value in  $\{Y, N, NA\}$ .  $R_Y^P, R_N^P$  and  $R_{NA}^P$  denote the sets of requests permitted, denied and not applicable by the policy  $P$ .
- $R_\Sigma = R_Y^P \cup R_N^P \cup R_{NA}^P$
- $\emptyset = R_Y^P \cap R_N^P$
- $\emptyset = R_Y^P \cap R_{NA}^P$
- $\emptyset = R_N^P \cap R_{NA}^P$
- Policy  $P$  is defined as the triple  $\langle R_Y^P, R_N^P, R_{NA}^P \rangle$ .
- $R_{NA}$  can always be mentioned as  $R_{NA} = R_\Sigma \setminus (R_Y \cup R_N)$

##### I.6.1.1.1. Basic Operators

###### Constants

- A *Permit* Policy:  $\mathcal{P}_Y \equiv \langle R_\Sigma, \emptyset, \emptyset \rangle$ . This policy permits everything.
- A *Deny* Policy:  $\mathcal{P}_N \equiv \langle \emptyset, R_\Sigma, \emptyset \rangle$ . This policy denies everything.

<sup>6</sup> $\text{dom}(a_i)$  is the set of all possible values  $v_i$  for the attribute  $a_i$  within the same domain.

**Addition** Addition combines two policies into a single one:

$$P_1 = P_2 + P_3 \iff \begin{cases} R_Y^{P_1} = R_Y^{P_2} \cup R_Y^{P_3} \\ R_N^{P_1} = (R_N^{P_2} \setminus R_Y^{P_3}) \cup (R_N^{P_3} \setminus R_Y^{P_2}) \end{cases}$$

This means that  $P_1$  is *Permit* for all attributes to which  $P_2$  and  $P_3$  are *Permit*.  $P_1$  is *Deny* for all attributes to which  $P_2$  and  $P_3$  are *Deny*.  $P_2$  and  $P_3$  may be in different domains.

**Addition Example** Consider the following two policies:

- $P_2 = \langle R_Y^{P_2}, R_N^{P_2}, R_{NA}^{P_2} \rangle$
- $P_3 = \langle R_Y^{P_3}, R_N^{P_3}, R_{NA}^{P_3} \rangle$

where

- $R_Y^{P_2} = \{a, b, c\}$
- $R_N^{P_2} = \{d, e, f\}$
- $R_{NA}^{P_2} = \{h, i\}$
- $R_Y^{P_3} = \{a, e, h\}$
- $R_N^{P_3} = \{b, f, i\}$
- $R_{NA}^{P_3} = \{c, d\}$
- $\Sigma = \{a, b, c, d, e, f, h, i\}$

$P_1 = P_2 + P_3$  equals  $\langle \{a, b, c\}, \{d, e, f\}, \{h, i\} \rangle + \langle \{a, e, h\}, \{b, f, i\}, \{c, d\} \rangle$  According to the specification of the *Addition* operator the resulting policy is:  $P_1 = \langle \{a, b, c, e, h\}, \{d, f, i\}, \emptyset \rangle$ .

**Intersection** The policy created through intersection (&) is applicable to all requests that have the same decisions for  $P_1$  and  $P_2$ :

$$P_1 = P_2 \& P_3 \iff \begin{cases} R_Y^{P_1} = R_Y^{P_2} \cap R_Y^{P_3} \\ R_N^{P_1} = R_N^{P_2} \cap R_N^{P_3} \end{cases}$$

**Intersection Example** Here we use the same two policies as in the Addition Example above.  $P_1 = P_2 \& P_3$  equals  $\langle \{a, b, c\}, \{d, e, f\}, \{h, i\} \rangle \& \langle \{a, e, h\}, \{b, f, i\}, \{c, d\} \rangle$ . According to the specification of the *Intersection* operator the resulting policy is:  $P_1 = \langle \{a\}, \{f\}, \{b, c, d, e, h, i\} \rangle$ .

**Negation** The negation ( $\neg$ ) inverts all decisions of a policy:

$$P_1 = \neg P_2 \iff \begin{cases} R_Y^{P_1} = R_N^{P_2} \\ R_N^{P_1} = R_Y^{P_2} \end{cases}$$

**Negation Example** Here we use  $P_2$  as it is defined in the Addition Example above.  $P_1 = \neg P_2$  equals  $\neg(\langle \{a, b, c\}, \{d, e, f\}, \{h, i\} \rangle)$  According to the specification of the *Negation* operator the resulting policy is:  $P_1 = \langle \{d, e, f\}, \{a, b, c\}, \{h, i\} \rangle$ .

**Domain Projection** The Domain Projection ( $\Pi_{dc}$ ) restricts a policy to the set of requests defined by a specific domain. This restriction is named *domain constraint* ( $dc$ ). A domain constraint  $dc$  has the following form:

$\{(a_1, range_1), (a_2, range_2), \dots, (a_n, range_n)\}$  where  $a_i$  are attribute names and  $range_i$  are values in  $dom(a_i)$ . A request  $r = \{(a_{r1}, v_{r1}), (a_{r2}, v_{r2}), \dots, (a_{rm}, v_{rm})\}$ , where  $a_{rj}$  are attribute names and  $v_{rj}$  are the corresponding attribute values, satisfies a domain constraint  $dc$  when  $\forall (a_{rj}, v_{rj}) \exists (a_i, range_i) \in dc$  so that  $a_{rj} = a_i$  and  $v_{rj} = range_i$  with  $(1 \leq i \leq n)$  and  $(1 \leq j \leq m)$ .

The semantics of  $\Pi_{dc}(P)$ :

$$P_1 = \Pi_{dc}(P_2) \iff \begin{cases} R_Y^{P_1} = \{r | r \in R_Y^{P_2} \text{ and } r \text{ satisfies } dc\} \\ R_N^{P_1} = \{r | r \in R_N^{P_2} \text{ and } r \text{ satisfies } dc\} \end{cases}$$

**Domain Projection Example** Here we use  $P_2$  as it is defined in the Addition Example above and assume that  $\{a, b, d, e\} \in A$  and  $\{c, f, h, i\} \in B$  where A and B denote two non-intersecting domains.

$$P_I = \Pi_A(P_1) = \langle \{a, b\}, \{d, e\}, \emptyset \rangle$$

The domain projection reduces the basic set  $\Sigma$ .

#### 1.6.1.1.2. Derived Operators

**Not applicable policy ( $\mathcal{P}_{NA}$ )** The not applicable policy constant  $\mathcal{P}_{NA}$  can be expressed as  $\mathcal{P}_Y \& \mathcal{P}_N$ .

$$\mathcal{P}_{NA} = \langle R_\Sigma, \emptyset, \emptyset \rangle \& \langle \emptyset, R_\Sigma, \emptyset \rangle = \langle \emptyset, \emptyset, R_\Sigma \rangle.$$

**Effect Projection ( $\Pi_Y$  and  $\Pi_N$ )** The effect projection  $P_1 = \Pi_Y(P_2)$  means that  $P_1$  is only applicable to request attributes to which  $P_2$  has the effect *Permit*.  $\Pi_Y(P_2) = P_2 \& P_Y$  and  $\Pi_N(P_2) = P_2 \& P_N$ .

**Effect Projection Example** Here we use  $P_2$  as it is defined in the Addition Example above.

$$\Pi_Y(P_2) = \langle \{a, b, c\}, \emptyset, \{d, e, f, h, i\} \rangle.$$

**Subtraction** The subtraction of two policies  $P_2 - P_3$  returns a policy that is applicable to requests applicable to  $P_2$  and not applicable to  $P_3$ .

$$P_2 - P_3 = (P_Y \& (\neg(\neg P_2 + P_3 + \neg P_3))) + (P_N \& (P_2 + P_3 + \neg P_3)).$$

**Subtraction Example** Here we use the same two policies as in the Addition Example above.  $P_2 - P_3 = \langle \{c\}, \{d\}, \{a, b, e, f, h, i\} \rangle$

**Precedence** Given two policies  $P_2$  and  $P_3$  the precedence operator ( $\triangleright$ ) returns the decision of  $P_2$  for all requests applicable to  $P_2$  and the decision of  $P_3$  to all remaining requests.

$$P_2 \triangleright P_3 = P_2 + (P_3 - P_2)$$

This operator can be used as a building block for conflict resolution, see section III.

**Precedence Example** Here we use the same two policies as in the Addition Example above.  $P_3 \triangleright P_2 = P_3 + (P_2 - P_3) = \langle \{a, c, e, h\}, \{b, d, f, i\}, \emptyset \rangle$

### 1.6.1.2. XACML Combining Algorithms

XACML defines 6 basic combining algorithms: (ordered-)permit-overrides, (ordered-)deny-overrides, first-applicable and only-one-applicable. These combining algorithms can be expressed in terms of FIA.

**1.6.1.2.1. Permit-overrides** The result of the combining algorithm is *Permit* if at least one policy evaluates to *Permit*. The result is *Deny* if at least one policy evaluates to *Deny* and no policy evaluates to *Permit*. This combining algorithm can be expressed as:

$$\sum_{i=0}^n P_i$$

**1.6.1.2.2. Deny-overrides** The result of the combining algorithm is *Deny* if at least one policy evaluates to *Deny*. The result is *Permit* if at least one policy evaluates to *Permit* and no policy evaluates to *Deny*. This combining algorithm can be expressed as:

$$\neg \sum_{i=0}^n \neg P_i$$

**1.6.1.2.3. First-applicable** The result of the combining algorithm is the result of the first policy that evaluates to *Permit* or *Deny*. This combining algorithm can be expressed as:

$$\triangleright_{i=1}^n P_i$$

where  $\triangleright_{i=1}^n P_i$  is equal to  $P_1 \triangleright P_2 \triangleright \dots \triangleright P_n$ .

**1.6.1.2.4. Only-one-applicable** The result of the combining algorithm is the result of the only applicable policy. If none or more than one policies are applicable than the result shall be *Not Applicable*.

$$(P_1 - P_2 - P_3 - \dots - P_n) + (P_2 - P_1 - P_3 - \dots - P_n) + \dots + (P_n - P_1 - P_2 - \dots - P_{n-1})$$

**1.6.1.2.5. Ordered Variants** The ordered variants of the combining algorithms, *ordered deny overrides algorithm* and *ordered permit overrides algorithm*, are neglected due to the fact that the behavior is exactly the same as with the unordered variants. The only difference between those is that the order of handling during evaluation is arbitrary in case of the unordered variants.

## I.7. Geometric Approach

### I.7.1. XACML Policy as N-Dimensional Rectangle

An XACML policy can be transformed into an  $n$ -dimensional rectangle in an  $n$ -dimensional space. Each attribute within the XACML *Target* forms one dimension. This transformation is bijective. Also the attributes can be mapped to any representation. For more details on the mapping of the attributes see Section II.3.

#### I.7.1.1. Basic Definitions

Every attribute that may occur describes a dimension in an  $n$ -dimensional space. All XACML matching functions that can appear in the *Target* are  $=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  and  $*$ -*match*. Where the  $*$ -*match* functions are either regex-functions, *RFC822Name-match* or *x500Name-match*. More details about how to map an XACML *Target* to an  $n$ -dimensional rectangle can be found in Section II.3 and is discussed by Oberholzer [16].

A policy is an  $n$ -dimensional rectangle in an  $n$ -dimensional space, created by  $x$  and  $y$ . Overlapping policies may occur. Such overlaps determine a conflict if the effect of the overlapping policies are not equal. See Section II.1 for further details.

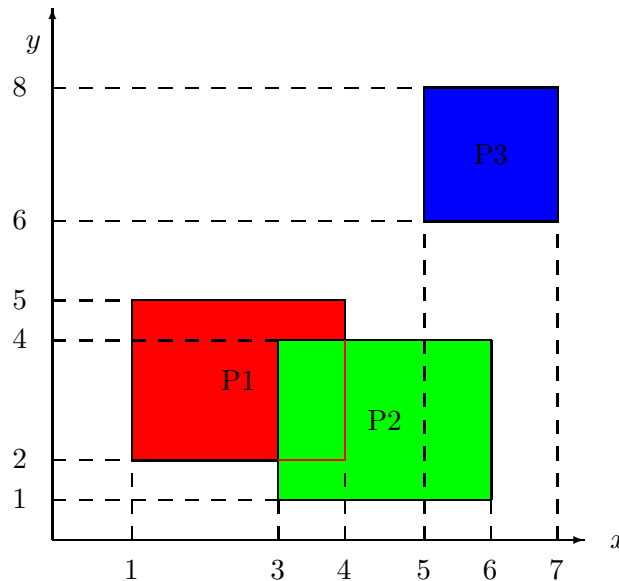


Figure I.3.: Policies in a 2-dimensional space

#### I.7.1.2. Example

For simplicity our example is in the 2-dimensional space.

As example we consider the following three policies (for a graphical illustration see Figure I.3).

- Policy 1:  $if((1 \leq x \leq 4) \text{ AND } (2 \leq y \leq 5)) \rightarrow \text{Permit}$
- Policy 2:  $if((3 \leq x \leq 6) \text{ AND } (1 \leq y \leq 4)) \rightarrow \text{Deny}$

- Policy 3:  $if((5 \leq x \leq 7) \text{ AND } (6 \leq y \leq 8)) \rightarrow Permit$

Note that these integers may be other types (e.g. time or boolean) that were mapped to integer for easier handling.

The overlap of Policy 1 and Policy 2 denotes a conflict regarding the definition in Section II.1. This conflict appears if a request contains the attribute  $x$  with a value between 3 and 4 and the attribute  $y$  with a value between 2 and 4.

## I.8. Algebraic Approach compared to the Geometric Approach

The *Algebraic Approach*, see Section I.6, is a good approach to describe operations (e.g. addition, subtraction, ...) on policies. The *Geometric Approach* on the other hand is qualified for the execution of such operations because there exist various well-established algorithms on  $n$ -dimensional rectangles.

Due to the fact that well-established algorithms for handling  $n$ -dimensional rectangles exist we decided to use this data structure as basis for our conflict detection.

## II. Conflict Detection

## II.1. Definition of Conflict

We define that a conflict of one or more XACML policies exists if those evaluate to different decisions (*Permit*, *Deny*) for the same request. For illustration consider the following four policies:

- Policy 1:  $if((1 \leq x \leq 4) \text{ AND } (2 \leq y \leq 5)) \rightarrow \textit{Permit}$
- Policy 2:  $if((3 \leq x \leq 6) \text{ AND } (1 \leq y \leq 4)) \rightarrow \textit{Permit}$
- Policy 3:  $if((1 \leq x \leq 4) \text{ AND } (5.5 \leq y \leq 7)) \rightarrow \textit{Deny}$
- Policy 4:  $if((3.5 \leq x \leq 6) \text{ AND } (3 \leq y \leq 6.5)) \rightarrow \textit{Deny}$

Figure II.1 and Figure II.2 show these four policies in the 2-dimensional space, created by the variables  $x$  and  $y$ . How to map policies to the  $n$ -dimensional space is described in Section II.3.

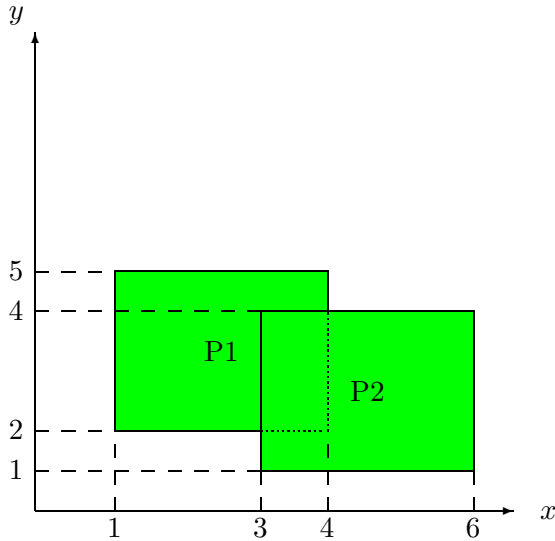


Figure II.1.: Permit policies

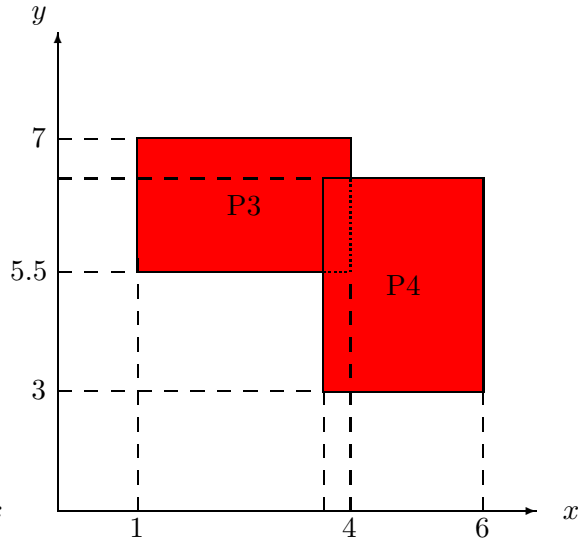


Figure II.2.: Deny policies

The reason to illustrate the *Permit* and *Deny* policies in two diagrams is that an overlap within the *Permit* or within the *Deny* policies is not a conflict. Only overlaps between *Permit* and *Deny* policies are conflicts.  $P^{Permit}$  and  $P^{Deny}$  are sets of policies containing all possible policies that evaluate to *Permit* or *Deny*, respectively.

For the above example  $P^{Permit} = \{P_1, P_2\}$  (see Figure II.3 on page 26 for a graphical illustration) and  $P^{Deny} = \{P_3, P_4\}$  (see Figure II.4 on page 26 for a graphical illustration).

A conflict exists if

$$\left( \bigcup_{P_i \in P^{Permit}} P_i \right) \cap \left( \bigcup_{P_i \in P^{Deny}} P_i \right) \neq \emptyset.$$

where the union and intersection operator on the sets operate on the points that the contained policies cover in the  $n$ -dimensional space. Therefore

$$C = \{x | x \in \mathbb{R}, x \in [3.5, 6]\} \cup \{y | y \in \mathbb{R}, y \in [3, 4]\}.$$

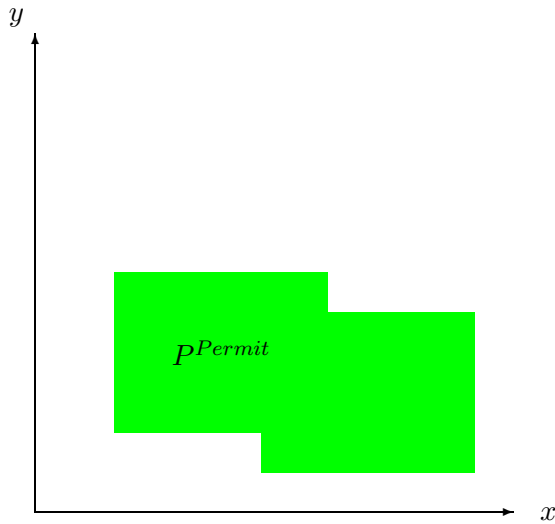


Figure II.3.:  $P^{Permit}$

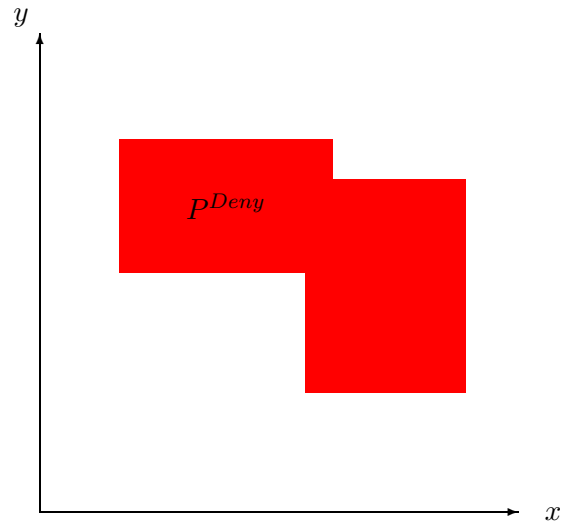


Figure II.4.:  $P^{Deny}$

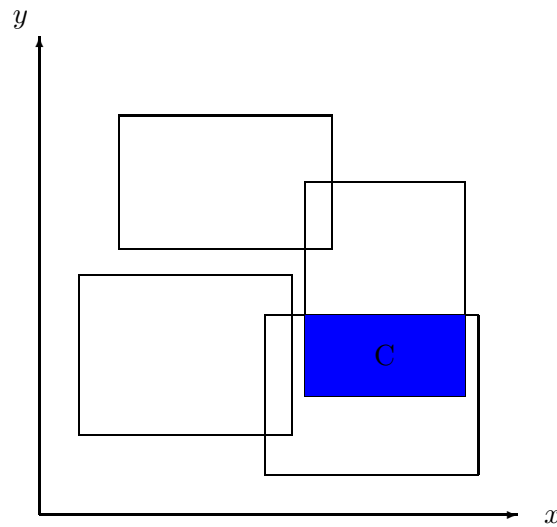


Figure II.5.: Conflict  $C$

This means that a conflict occurs if a request arrives with an  $x$  value between 3.5 and 6 and a  $y$  value between 3 and 4. The conflicting area can be seen in Figure II.5.

## II.2. Our Approach

Our conflict detection algorithms only detect conflicts among XACML *Targets*. We do not consider the *Condition* of the XACML *Rule* elements. Our conflict detection algorithms do only find conflicts among *Targets*.

Our approach for finding conflicts among XACML *Targets* is to map every *Virtual Target* to the  $n$ -dimensional space and to find all overlaps of policies with different *Effects*. With this approach every policy can be seen as a set of attributes and a conflict exists between two or more policies having a non-empty intersection in all dimensions. See Section II.1 for further details. The mapping of different matching functions to the  $n$ -dimensional space is described in Section II.3 and the approach of detecting conflicts is described in Section II.4. The reason that the matching function is described, and not only the data type, is because the matching function reveals how a specific data type is mapped.

In this thesis we propose algorithms that detect and resolve conflicts among policies with different *Effects* (*deny*, *permit*). Due to the fact that the algorithms are very generic they can be used for any kind of conflicts. Only the criteria has to be changed what a conflict is. It is easily imaginable that another application would be to find conflicts of *Obligations*. Maybe it is not always preferable that all *Obligations* of all applicable policies are returned.

## II.3. Mapping XACML Target to the N-Dimensional Space

To be able to detect conflicts among policies (here with *policy* we mean the *Target* of a *Rule* element), those policies must be transformed into  $n$ -dimensional rectangles. Every *Virtual Target* forms an  $n$ -dimensional rectangle. This structure has the advantage that a lot of well-established algorithms can be used to detect conflicts.

Every dimension in the  $n$ -dimensional space is formed by an attribute from the *Target*. The number of dimensions is equal to the number of attributes that exists among the vocabulary containing all possible attribute names and their corresponding domain. How the attribute value of the attributes within a policy (or *Virtual Target*, respectively) is mapped to the dimension of the corresponding attribute is determined by a comparison function.

The basic idea is to map all the attribute values to the  $n$ -dimensional space by dualizing them into either an integer, a double, a string or a finite automaton. All data types can easily be mapped to one of these four types. The reason to reduce the number of XACML data types to four is to ease the handling of them and make it more uniform [16]. This because XACML specifies 16 data types and it is a lot easier to limit to a few data types.

The mapping to a finite automaton is a special case and must be undertaken when there is a certain attribute within any policy that is matched with a *\*-match* function. In this case the mapping may look very scattered, see Figure II.6 for example. This means that the *Target* of a policy does not form a range within this particular dimension and this cannot be mapped very easily. Therefore we use finite automata to handle this special case, see Section II.3.4.

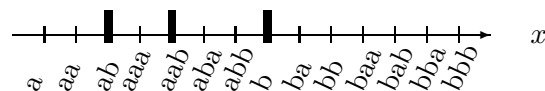


Figure II.6.: Regular expression  $a^*b$ .

### II.3.1. Attribute Types mapped to Integer

In this section all data types that are mapped to an integer are listed. Additionally, it is described how this mapping may be done.

<http://www.w3.org/2001/XMLSchema#integer>

No mapping is needed because this type is already an integer.

<http://www.w3.org/2001/XMLSchema#boolean>

This data type has only two possible states: *true* and *false*. The mapping is *false*  $\rightarrow$  0 and *true*  $\rightarrow$  1.

### **<http://www.w3.org/2001/XMLSchema#time>**

First this data type is normalized to the GMT<sup>1</sup> time zone. The integer representation then is the milliseconds since 00:00 GMT.

### **<http://www.w3.org/2001/XMLSchema#date>**

The integer representation is the milliseconds since 01/01/1970 00:00:00 GMT. A date before 01/01/1970 is represented as a negative integer value. The time component is always 00:00:00 to only consider the date part.

### **<http://www.w3.org/2001/XMLSchema#dateTime>**

The integer representation is the milliseconds since 01/01/1970 00:00:00 GMT. A dateTime before 01/01/1970 00:00:00 GMT is represented as a negative integer value.

### **<urn:oasis:names:tc:xacml:2.0:data-type:dayTimeDuration>**

The integer representation is the number of seconds of this duration.

### **<urn:oasis:names:tc:xacml:2.0:data-type:yearMonthDuration>**

The integer representation is the number of months of this duration.

## **II.3.2. Attribute Types mapped to Double**

In this section all data types that are mapped to a double are listed. Additionally it is described how this mapping may be done. The only data type that fits into this category is the <http://www.w3.org/2001/XMLSchema#double> data type.

### **<http://www.w3.org/2001/XMLSchema#double>**

No mapping is needed because this is already a double.

## **II.3.3. Attribute Types mapped to String**

In this section all data types that are mapped to a string are listed. The representation that these data types have in their XML form is taken.

If an attribute has, in any policy, a match with a *\*-match* function the data type cannot simply be mapped to a string. In this case the string representation must further be transformed into a finite automaton. See section II.3.4 for further information. The following data types must be transformed into their string representation:

- <http://www.w3.org/2001/XMLSchema#string>
- <http://www.w3.org/2001/XMLSchema#anyURI>
- <http://www.w3.org/2001/XMLSchema#hexBinary>
- <http://www.w3.org/2001/XMLSchema#base64Binary>

---

<sup>1</sup>GMT: Greenwich Mean Time

- urn:oasis:names:tc:xacml:1.0:data-type:x500Name<sup>2</sup>
- urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name

### II.3.4. Attribute Types mapped to Finite Automata

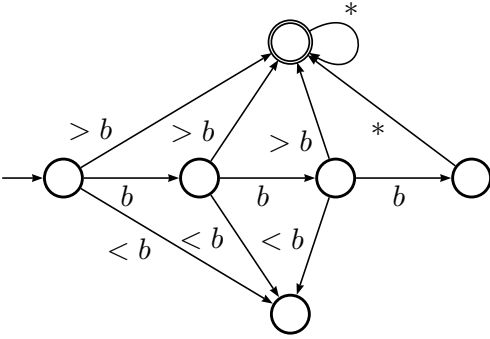
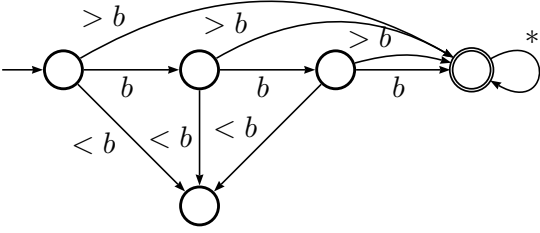
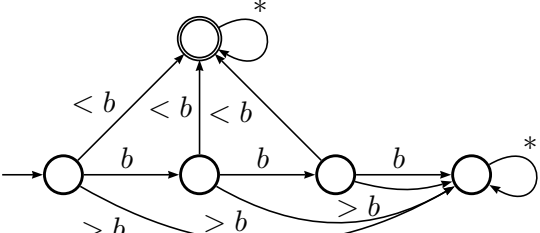
As described above (see Figure II.6 on page 28) the reason to map a *\*-match* function to a finite automaton is because the mapping may look very scattered. This means that in this dimension (i.e. attribute) the range where the policy is applicable is not coherent. In such a case it is not possible to detect intersections with our proposed algorithms. We can use the theory of finite automata to find intersections among regular expressions very easily. The following data types must be transformed into their string representation **when there exists at least one *\*-match* function** among all policies that contain the same attribute.

- http://www.w3.org/2001/XMLSchema#string
- http://www.w3.org/2001/XMLSchema#anyURI
- urn:oasis:names:tc:xacml:1.0:data-type:x500Name<sup>2</sup>
- urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name
- urn:oasis:names:tc:xacml:2.0:data-type:ipAddress
- urn:oasis:names:tc:xacml:2.0:data-type:dnsName

To be able to check if there is an intersection when a data type is mapped once with a *\*-match* function and at least once with a comparison function, the comparison functions must also be transformed into finite automata. How this mapping is done is described in Table II.1. The mapping is only described for strings because all data types are mapped to a string before matching with a regular expression. The *\** character means *any character*, the *>* operator means all characters greater than the given one and *<* means all characters smaller than the given one.

<p>equals (=)</p>	<p>A finite automaton must be constructed that accepts only the given string. For example we map “= <i>bbb</i>” to a finite automaton.</p>
-------------------	--

<sup>2</sup>x500Name could also be handled as a multi-valued type.

<p>greater-than (<math>&gt;</math>)</p>	<p>A finite automaton must be constructed that accepts only strings that are greater than the given string. The finite automaton for "<math>&gt; bbb</math>" is mapped the following way:</p>  <p>The diagram shows a finite automaton with five states. The start state is the leftmost circle. Transitions are: start to state 2 on 'b', start to state 4 on '&gt; b', state 2 to state 3 on 'b', state 2 to state 5 on '&lt; b', state 3 to state 4 on 'b', state 3 to state 5 on '&lt; b', state 4 to state 5 on '&lt; b', state 4 to state 1 on '&gt; b', state 5 to state 1 on '&gt; b', state 1 to state 1 on '*', state 4 to state 1 on '*', and state 5 to state 1 on '*'. State 1 is the final state.</p>
<p>greater-than-or-equal (<math>\geq</math>)</p>	<p>A finite automaton must be constructed that accepts strings that are greater or equal than the given string. The finite automaton for "<math>\geq bbb</math>" is mapped the following way:</p>  <p>The diagram shows a finite automaton with five states. The start state is the leftmost circle. Transitions are: start to state 2 on 'b', start to state 4 on '&gt; b', state 2 to state 3 on 'b', state 2 to state 5 on '&lt; b', state 3 to state 4 on 'b', state 3 to state 5 on '&lt; b', state 4 to state 5 on '&lt; b', state 4 to state 1 on '&gt; b', state 5 to state 1 on '&gt; b', state 1 to state 1 on '*', state 4 to state 1 on '*', and state 5 to state 1 on '*'. State 1 is the final state.</p>
<p>lesser-than (<math>&lt;</math>)</p>	<p>A finite automaton must be constructed that accepts strings that are lesser than the given string. The finite automaton for "<math>&lt; bbb</math>" is mapped the following way:</p>  <p>The diagram shows a finite automaton with five states. The start state is the leftmost circle. Transitions are: start to state 2 on 'b', start to state 4 on '&lt; b', state 2 to state 3 on 'b', state 2 to state 5 on '&gt; b', state 3 to state 4 on 'b', state 3 to state 5 on '&gt; b', state 4 to state 5 on '&gt; b', state 4 to state 1 on '&lt; b', state 5 to state 1 on '&lt; b', state 1 to state 1 on '*', state 4 to state 1 on '*', and state 5 to state 1 on '*'. State 1 is the final state.</p>

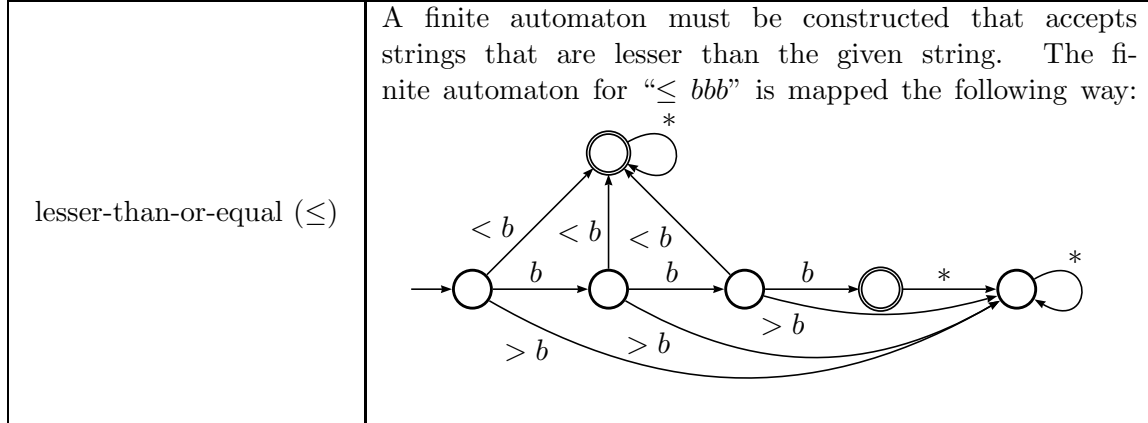


Table II.1.: *string*-\* matching functions as finite automata.

The mapping of regular expressions into a finite automata can be made according to the *Thompson Construction* [20, 18]. The *Thompson Construction* creates an  $\epsilon$ -NFA. To transform the  $\epsilon$ -NFA into an  $\epsilon$ -free-NFA takes  $O(n^3)$  time [10]. To avoid this Hagenah and Muscholl [7] described an algorithm that directly creates an  $\epsilon$ -free-NFA from a regular expression. See Section II.4.2.3 for further details. An example of a mapping with the *Thompson Construction* is in Table II.2 on page 34.

A specialty among the \*-match functions are the *urn:oasis:names:tc:xacml:1.0:function:x500Name-match* and *urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match*. To also be able to map these functions those can be transformed into a regular expression [16] and then they can be handled like all the other regular expressions as described above.

### Transforming of *urn:oasis:names:tc:xacml:1.0:function:x500Name-match*

An x500Name may appear in two different forms. Either in RDN form or in ASN.1 form. To be able to match two x500Names its is required to make the values uniform. This is done in 2 steps:

1. Normalize the value according to RFC2253 [22] into their string representation.
2. Re-order the attribute/value pairs in ascending order when compared as octet strings according to ITU-T X.690 [21].

After this unification the strings may be compared using the rules in RFC 3280 [11, Section 4.1.2.4].

To be able to detect overlaps within x500Names those shall be transformed into a regular expression. This is done the following way:  $regex = .*attr_1, attr_2, \dots, attr_n$ .

## Transforming of urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match

An rfc822Name is built up on a local, case-sensitive<sup>3</sup>, and a domain, case-insensitive, part. To avoid problems with internationalized domain names the *ToASCII* algorithm described in RFC3490 [6] shall be applied to the domain-part of all rfc822Names that have an internationalized domain-part. The reason is that there are letters where no round-trip is possible ( $X \neq (X.toLowerCase()).toUpperCase()$ ), e.g. in Quebec the standard uppercase equivalent of “è” is “È”, while in metropolitan France it is more commonly “E”.

The regular expression is built with first applying the *ToASCII* transformation and second the *toLowerCase()* transformation on all remaining ASCII characters.

**foo@bar.ch** This rfc822Name is mapped to  $foo@[bB][Aa[Rr]\.[Cc][Hh]$ . This matches only the specific rfc822Name *foo@bar.ch*.

**bar.ch** This rfc822Name is mapped to  $\backslash S^+@[bB][Aa[Rr]\.[Cc][Hh]$ . This matches all rfc822Names under the domain *bar.ch*, but not any sub-domain of *bar.ch*.

**.bar.ch** This rfc822Name is mapped to  $\backslash S^+@(\.[* \.])?[bB][Aa[Rr]\.[Cc][Hh]$ . This matches all rfc822Names under the domain *bar.ch*, it also matches all sub-domains of *bar.ch*.

**foo@müller.ch** This rfc822Name is mapped to  $foo@xn--mller-kva\ch$ . This matches only the specific rfc822Name *foo@müller.ch*. Note that the *ToASCII* algorithm also performs a *toLowerCase()* operation.

---

<sup>3</sup>Many mail systems handle the local part as case-insensitive. This does not conform to the specification [3, 13] and is therefore not respected.

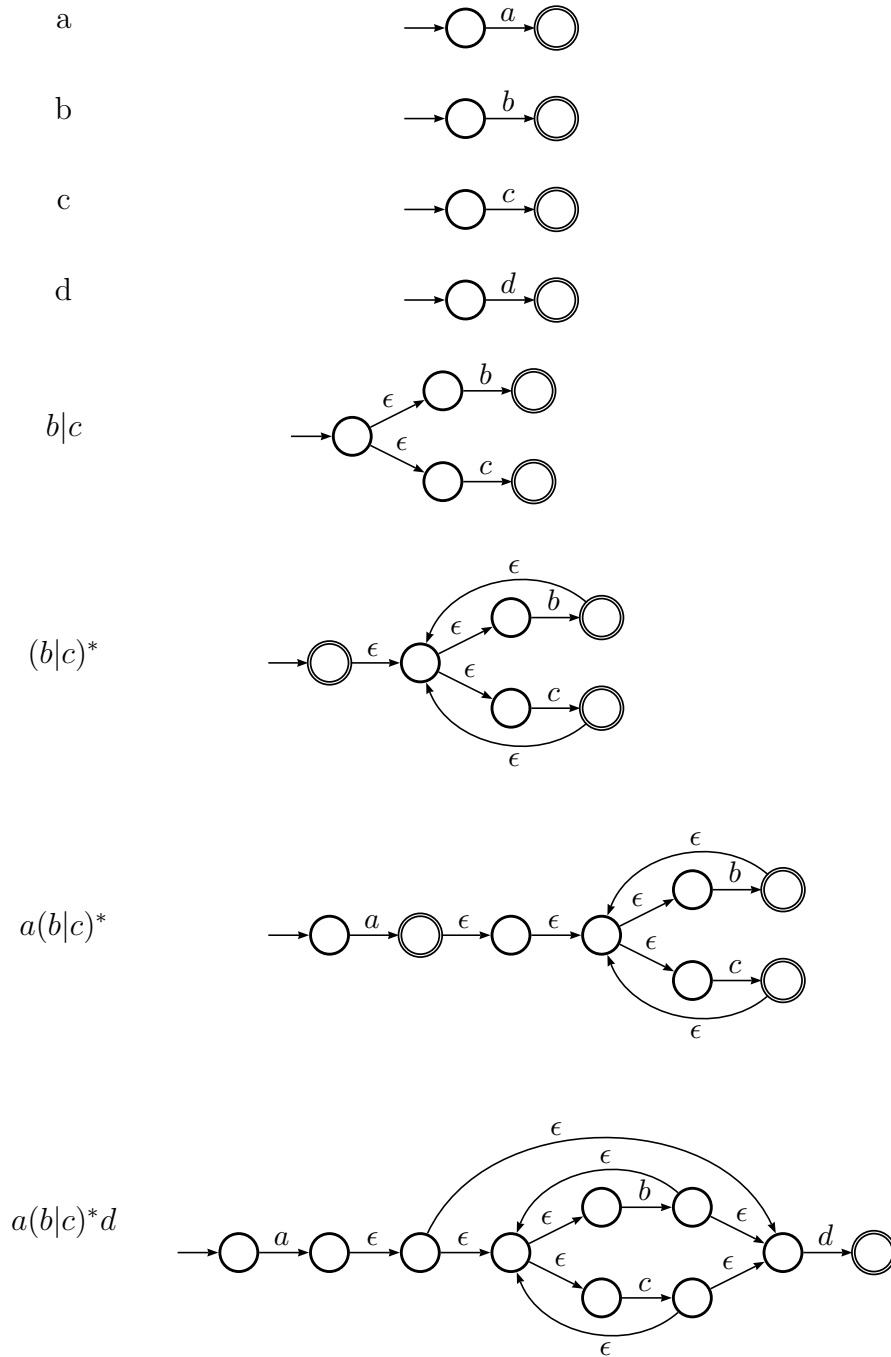


Table II.2.: Thompson construction of the regular expression  $a(b|c)^*d$ .

## II.4. Conflict Detection Algorithm

Our *Conflict Detection Algorithm* of policies (here a policy is meant as the *Target* element of a *Rule*) consists of various parts. The main idea is to represent the policies in the  $n$ -dimensional space as  $n$ -dimensional rectangles. The algorithm then runs through each dimension and detects intersections among the policies. How this detection is performed depends on the data type of the attribute in each dimension and the ratio of permit/deny policies. The *Conflict Detection Algorithm* has the following steps:

1. Map all policies to the  $n$ -dimensional space
2. Determine start- and endpoint of the range every policy covers
3. For each dimension
  - a) Determine all pairwise intersections
    - i. Either with the *Plane Sweep Algorithm*,
    - ii. the *AABB - AABB Intersection Algorithm*,
    - iii. or (with regular expressions) the *Regex Intersection Algorithm*
  - b) Prune all policies that cannot conflict another one
4. Report all pairwise conflicts

In the following sections each step is explained in detail. The first step (Map all policies to the  $n$ -dimensional space) was already explained in Section II.3.

### II.4.1. Determining Start- and End-Point of the range a policy covers

Due to our algorithm it is necessary to determine the start- and end-point of the range a policy covers. This must be done for every dimension of the  $n$ -dimensional space the policy is in. The range that a policy covers in a specific dimension is defined by the comparison function of a specific attribute. There are three possibilities how such a range can look like:

- **A closed range:**  $[x; y]$ ,  $(x; y)$ ,  $(x; y]$  or  $[x; y)$ .  
E.g.  $x > 5$  AND  $x \leq 10 \rightarrow (5; 10]$ .
- **An open range:**  $(\infty; y]$ . An open range has the same structure as the closed range but either the upper bound or the lower bound is  $\pm\infty$ . The  $\pm\infty$  bound can only be exclusive (round bracket).  
E.g.  $x \leq 2 \rightarrow (-\infty; 2]$ .
- **A single point:**  $[x; x]$ . A single point means that no range function is applied but only an equality function. In terms of a range this means that the lower and the upper bound are equal.  
E.g.  $x = 10 \rightarrow [10; 10]$ .

Regular expressions cannot be expressed in terms of ranges. Regular expressions must be transformed into finite automata.

**Note:** If an attribute is not present in a policy  $A$  but in at least one other policy  $B$  then this attribute must match all possible values  $(-\infty; +\infty)$  for policy  $A$ .  $\pm\infty$  are markers for the most lower and upper bound, respectively.

## II.4.2. Determine all pairwise intersections

All pairwise conflicts among all policies must be determined to be able to resolve all conflicts. It is sufficient to determine pairwise conflicts only. The resolution of conflicts of three or more overlapping policies can be reduced to the pairwise conflict resolutions (see Chapter III).

We suggest three algorithms for determining conflicts, each having advantages for particular problem statements. A detailed evaluation of the three algorithms

- Plane Sweep Algorithm
- AABB - AABB Intersection Algorithm
- Regex Intersection Algorithm

is given in section II.4.6.2.

Our approach to find all pairwise conflicts among all policies is to run through each dimension and find all intersecting policies within each particular dimension. Two policies have a conflict if they intersect in every dimension and have a different *Effect* (i.e. *Permit* or *Deny*). That the policies must intersect in every dimension is constitutional as easily can be seen in Figure II.7. Here the two policies  $P1$  and  $P2$  intersect in dimension  $x_1$  but not in dimension  $x_2$ , so they have no conflict. If there is a policy that has no intersection

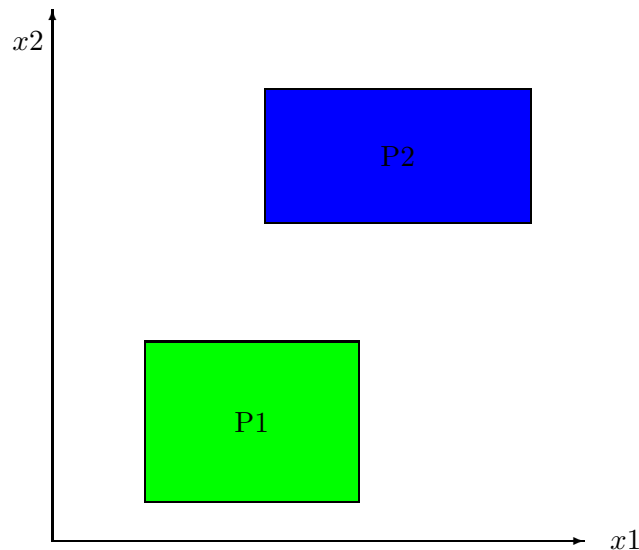


Figure II.7.: Two non-conflicting policies

with any other policy it can be excluded for all upcoming dimensions and thus speed up the further steps.

To be able to find intersections among the policies within a particular dimension the policies must be projected onto a single dimension. The reason is that the algorithms run dimension by dimension to detect conflicts. Figure II.8 on page 37 shows how the projection of two policies in the 2-dimensional space can look like when projecting onto a one single dimension.

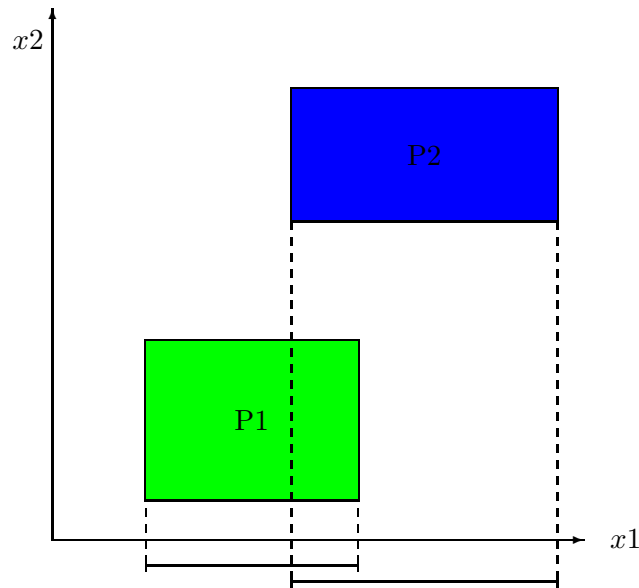


Figure II.8.: Projection of two policies.

#### II.4.2.1. Plane Sweep Algorithm

The *Plane Sweep Algorithm* has its origin in applications like digital maps. There the *Plane Sweep Algorithm* is used to determine whether streets and rivers overlap. In practice, this would show if a particular street has a bridge over a particular river. Further details regarding the *Plane Sweep Algorithm* are discussed by de Berg et al. [4].

We slightly adapted the algorithm to be able to determine if policies intersect. We do this by sweeping a plane through the dimension under inspection. Before the plane is swept through the dimension all policies are projected onto this dimension and form ranges (lines).

Our algorithm then walks through three steps to determine all intersections within this particular dimension:

1. Sort
2. Sweep the Plane
3. Report Intersections

**Sort** The start- and end-points of the policies are sorted by the order given by the data type in the XACML 2.0 specification. The policies must be sorted due to the fact that the *Plane Sweep Algorithm* does only work on a sorted set of start- and end-points. The sorting can be narrowed to a runtime behavior of  $O(n * \log(n))$  [15] with algorithms like *Introspective Sort*.

Another approach could be to use *Selection Sort* that runs in  $O(n^2)$  but has the advantage that the next steps can run simultaneously while sorting. Here we do not investigate this approach further.

**Sweep the Plane** Here we sweep a plane through the dimension under inspection and put the policy into a set  $S$  when the startpoint of this policy is reached. Sweeping a plane

means jumping through the sorted list of start- and end-points. The policy is removed from  $S$  when the corresponding end-point is reached. Two or more policies in  $S$  at the same time denote an intersection between those.

**Report intersections** Report all pairwise intersections that were detected.

**Example** Figure II.9 on page 39 shows a simple example of a plane sweeping through a dimension named  $x_1$ . In Subfigure II.9(a) the set  $S$  is empty due to the fact that the sweep plane does not intersect with any policy. When moving the sweep plane further it reaches a point where it looks like Subfigure II.9(b) where  $P1$  is put into the set  $S$ . In the next step, Subfigure II.9(c), two policies ( $P1$  and  $P2$ ) are intersected by the sweep plane. At this point more than one policy is in set  $S$  and this denotes an intersection and maybe a conflict. When moving further, Subfigure II.9(d),  $P1$  is removed from  $S$  and only  $P2$  remains in the set  $S$ . The last step, Subfigure II.9(e), before the plane sweep ends is removing  $P2$  from  $S$  as well.

The result of this example is that  $P1$  and  $P2$  intersect in dimension  $x_1$ . This means that if these two policies also intersect in all other dimensions they have a conflict.

#### II.4.2.2. AABB - AABB Intersection Algorithm

The *AABB - AABB Intersection Algorithm* has its origin in applications like e.g. collision detection of objects, e.g. human beings, in games. AABB is the acronym of *Axis Aligned Bounding Box*. The intent is to put an AABB around a more complex polygon, e.g. the human being, to determine if the objects may collide. If the AABBs collide, more complex algorithms can be applied to find out if the concrete polygons collide as well [5]. This approach makes the collision detection much faster. In our case the concrete polygons are already AABBs because the *Virtual Targets* of XACML policies always form  $n$ -dimensional rectangles.

The algorithm walks through three steps to determine all intersections within a particular dimension:

1. Create bounding sets
2. Check for intersections of AABBs
3. Report Intersections

**Create bounding sets** In this step two sets  $L$  and  $U$  are created from the set  $P$  that contains the policies.  $L$  contains all lower and  $U$  all upper bounds of the policies. The two sets are disjunct. If a policy is projected to one point in a particular dimension this point is the upper and lower bound at the same time. This is the only case where the same bound can be in both sets.

Further the set  $L$  can be split into two disjunct sets  $L^{Permit}$ , containing all lower bounds of all permit policies, and  $L^{Deny}$ , containing all lower bounds of all deny policies, where  $L = L^{Permit} \cup L^{Deny}$ . The same holds for the set  $U$  where  $U = U^{Permit} \cup U^{Deny}$ .

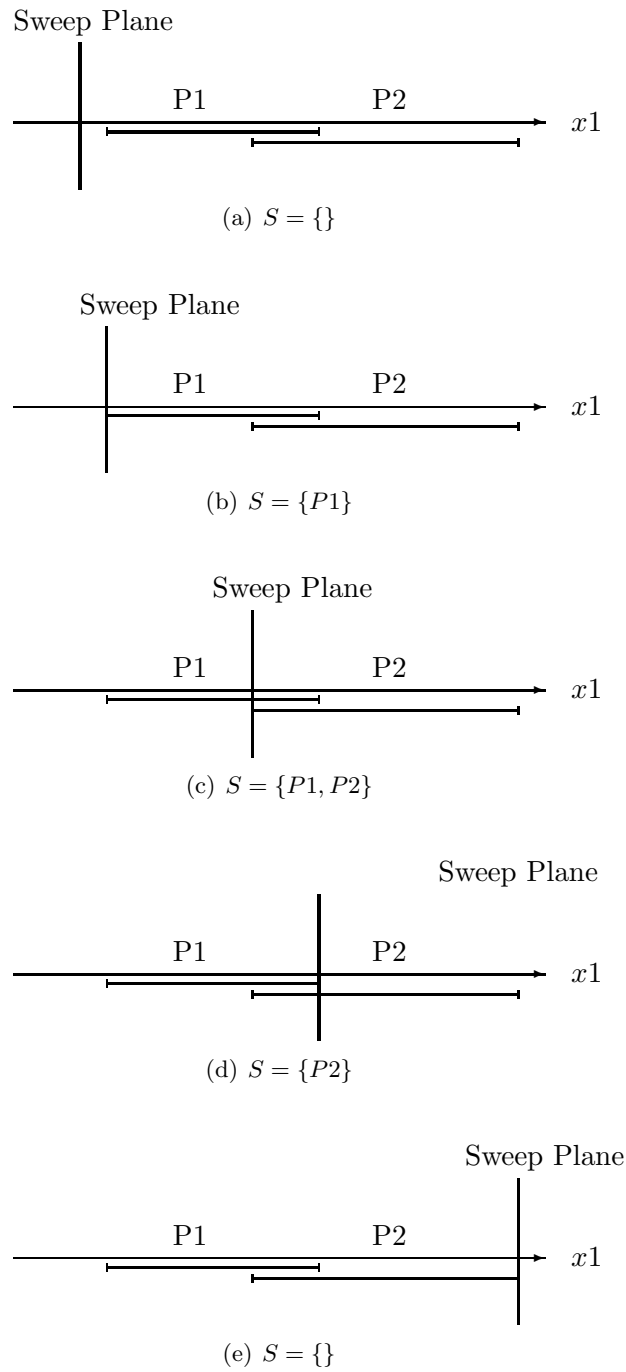


Figure II.9.: Plane sweeps through dimension  $x_1$

**Check for intersections of AABBs** Here the elements from  $L$  and  $U$  are checked by the following expression:

$$u_i < l_j \vee l_i > u_j : l \in L^x; u \in U^y; x, y \in \{Permit, Deny\}$$

where  $x, y$  are distinct elements of  $\{Permit, Deny\}$ . An intersection is present if the expression evaluates to *false*. Regard that also the borders have to be checked. If the range borders built by the elements from  $U$  and  $L$  both are built up from either an *equals*, *greater-than-or-equal* or *lesser-than-or-equal* comparison function an intersection is also present.

**Report Intersections** Report all pairwise intersections that were detected.

**Example** In this example we build up the sets and check two policies for intersection. Consider two policies

$$P1 : \text{if}(20 \leq x \leq 25) \rightarrow Permit$$

$$P2 : \text{if}(23 \leq x \leq 30) \rightarrow Deny$$

The sets  $P$ ,  $L$  and  $U$  of these two policies are:

$$P = \{P1, P2\}$$

$$U = \{25, 30\} = U^{Permit} \cup U^{Deny}, U^{Permit} = \{25\}, U^{Deny} = \{30\}$$

$$L = \{20, 23\} = L^{Permit} \cup L^{Deny}, L^{Permit} = \{20\}, L^{Deny} = \{23\}$$

From this the following expression ( $u_i < l_j \vee l_i > u_j : l \in L^x; u \in U^y; x, y \in \{Permit, Deny\}$ ) results:

$$25 < 23 \vee 20 > 30 \rightarrow false$$

Due to the fact that the expression evaluates to *false* the policies  $P1$  and  $P2$  intersect.

Another example, consider the two policies

$$P1 : \text{if}(20 \leq x \leq 25) \rightarrow Permit$$

$$P2 : \text{if}(26 \leq x \leq 30) \rightarrow Deny$$

The sets  $P$ ,  $L$  and  $U$  of these two policies are:

$$P = \{P1, P2\}$$

$$U = \{25, 30\} = U^{Permit} \cup U^{Deny}, U^{Permit} = \{25\}, U^{Deny} = \{30\}$$

$$L = \{20, 26\} = L^{Permit} \cup L^{Deny}, L^{Permit} = \{20\}, L^{Deny} = \{26\}$$

From this the following expression  $u_i < l_j \vee l_i > u_j : l \in L^x; u \in U^y; x, y \in \{Permit, Deny\}$  results:

$$25 < 26 \vee 20 > 30 \rightarrow true$$

Due to the fact that the expression evaluates to *true* the policies  $P1$  and  $P2$  do not intersect.

### II.4.2.3. Regex Intersection Algorithm

The *Regex Intersection Algorithm* is used to detect intersections between regular expressions. A regular expression accepts a set of strings. With the help of the theory of finite automata it is possible to determine whether two regular expressions intersect.

Two regular expressions  $R_1$  and  $R_2$  intersect if

$$R_1 \cap R_2 \neq \emptyset.$$

It is a fact that regular expressions are equal to a finite automata in their power of expressiveness [18]. This equality can be used because the intersection of two finite automata is easy to determine. The algorithm to detect if two finite automata<sup>4</sup> ( $F_1$  and  $F_2$ ) intersect has the following steps

1. Create a new finite automaton  $F$ .  $F$  is the *Cartesian Product* of  $F_1$  and  $F_2$ .
2. The *start state* of  $F$  is the state that holds the *start states* of  $F_1$  and  $F_2$ .
3. The *accepting states* of  $F$  are those states that are *accepting states* for  $F_1$  and  $F_2$ .
4. For each state of  $F$ 
  - a) Iterate over all elements of the alphabet  $\Sigma$ .
  - b) Start on both original finite automata  $F_1$  and  $F_2$  simultaneously and enter the element of  $\Sigma$ .
  - c) Make a connection in  $F$  to the state that holds both target states that were reached by entering the element of  $\Sigma$  in  $F_1$  and  $F_2$ .
5. Make a *Depth-First Search* starting at the *start state* of  $F$ . Keep all paths that end in a *accepting state* of  $F$ .

If at least one path to an *accepting state* exists, the finite automata, and therefore the regular expressions, have an intersection<sup>5</sup>.

**Example** As an example we want to check if the following two regular expressions intersect:

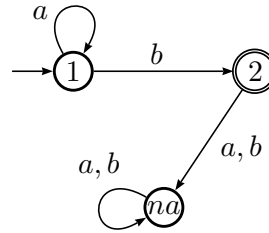
- $R_1 = a * b$
- $R_2 = bb*$

The example of the algorithm can be seen in Table II.3.

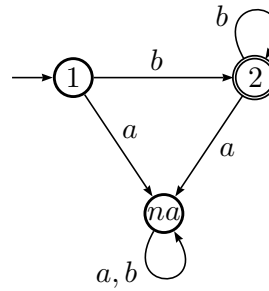
<sup>4</sup>It is important that the finite automata are complete, this means that they have an *NA* state.

<sup>5</sup>This intersection can easily be transformed into a regular expression with removing all elements that are not part of the path from  $F$  and transform this new finite automaton back into a regular expression.

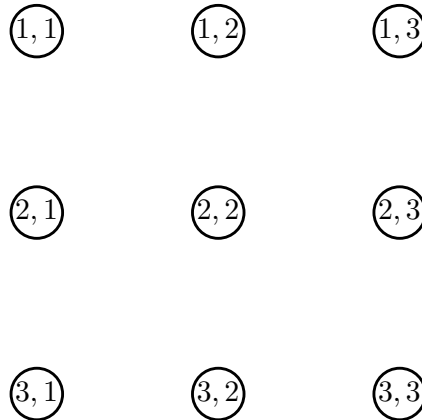
$$F_1 = R_1 = a * b$$



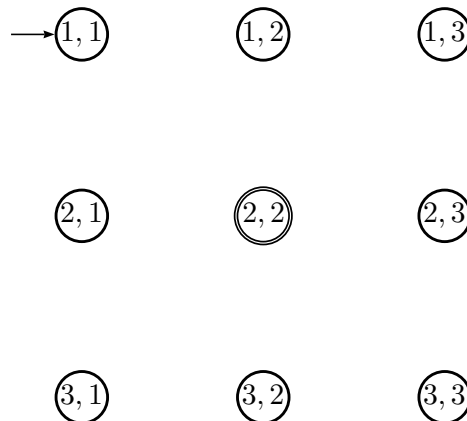
$$F_2 = R_2 = bb*$$



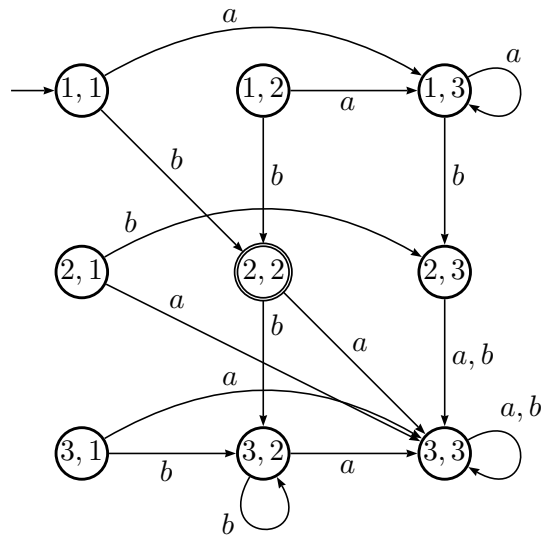
States of  $F$  (3 stands for  $na$ )



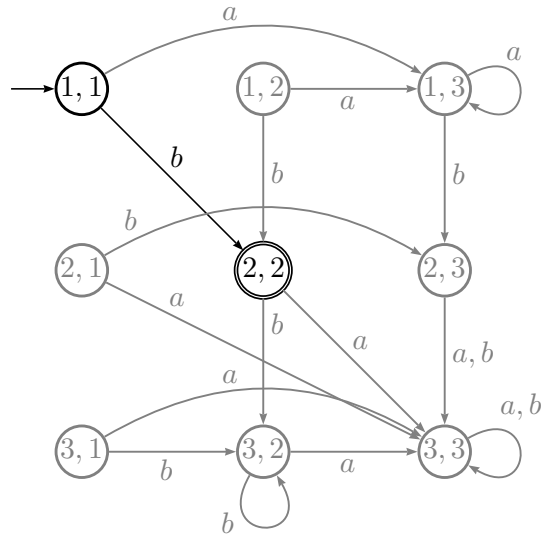
Start state and accepting states



Connections between the states



Remove all paths that do not end in an *accepting state* (the gray states and transitions are removed)



The result is a finite automaton that accepts only the string *b*. This means that the two regular expressions intersect. The intersection is the string *b*.

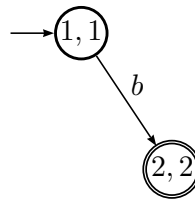


Table II.3.: Example of the *Regex Intersection Algorithm*.

### II.4.3. Pruning all non-intersecting policies

After all intersections within a dimension are detected the policies that do not intersect with any other can be removed from the global list of possible conflicting policies. The policy does not need to be checked again in other dimensions because it can already be

excluded.

The reason is, as described above (see Figure II.7 on page 36), a conflicting policy must have intersections with another policy in each dimension.

A possible approach to handle the pruning of policies is described in Section II.4.5.2.

#### II.4.4. Report all pairwise conflicts

When the intersections in the last dimension are detected then these intersections can be reported as conflicts. It is only logical that intersections that are found in the last dimension are intersections in all preceding dimensions as well.

The conflicts now can be resolved. See chapter III for details.

#### II.4.5. Implementation Hints - Data Structures

Hereafter we describe two data structures that cause an implementation to be much faster. The *Intersection-Matrix* can be used to track the intersections through the dimensions. It has the advantage that after the last dimension it is immediately clear which policies do conflict. There is no further checking needed.

The second data structure is a *Global Policy List*. A pruned policy does not need to be checked in upcoming dimensions again because it is already clear that this policy cannot conflict with any other. The *Conflict Detection Algorithms* shall take the policies from such a *Global Policy List* to only check possibly conflicting policies with the others.

##### II.4.5.1. Intersection-Matrix

When implementing the algorithm it is very important to use an appropriate data structure. When simply collecting the conflicts in a list and comparing all those lists in the end to see which intersections occur in every dimension, this would take  $O(\binom{n}{2}^d)$  operations, where  $d$  is the number of dimensions (i.e. attributes),  $n$  the number of policies and  $\binom{n}{2}$  denotes the maximum number of pairwise intersections within one dimension.

This can be avoided when using a global  $n \times (d + 1)$  matrix with the following characteristics:

- The first row contains all attributes ordered by the attribute that forms the first dimension.
- All rows, except of the first one, of the matrix contain lists that contain the attributes that intersect with the attribute in the corresponding first row.
- For all lists (except of the first two rows) the elements in the list must be equal or a subset of the preceding list.

An example of this matrix could look like this:

$$\begin{bmatrix} A & B & C & D & E & F \\ \{\} & \{C, F, E, D\} & \{F, E, D\} & \{E, F\} & \{F\} & \{\} \\ \{\} & \{D, C\} & \{\} & \{E\} & \{\} & \{\} \end{bmatrix}$$

The idea behind this data structure is that the first row is built and inserted into the matrix. Afterwards a *Conflict Detection Algorithm* walks through the first dimension and

adds all intersections to the corresponding lists in the second row. Then a *Conflict Detection Algorithm* walks through the remaining dimensions and for each dimension all intersections that are already present in the previous dimension are added to the list. In the end all intersections that are left in the last dimension are conflicts because this means that this intersection is present in all dimensions.

It is very important to insert the elements in the order of the first dimension into the lists. This means if an intersection between two policies is found in the second or a higher dimension it is required to sort these two policies in the order of the first dimension.

As example consider the three policies  $A, B, C$ . Their order in the first dimension is  $A \rightarrow B \rightarrow C$ . Now if in the second dimension an intersection between  $C, B$  is found. Then  $C, B$  must be sorted according to the first dimension, this is  $B, C$ . As a result  $C$  is added to the list of  $B$  in the second dimension and not vice versa.

As a result of this the lists in the last column are always empty.

**Example** Here we show a small example. We look at the three policies in Figure II.10.

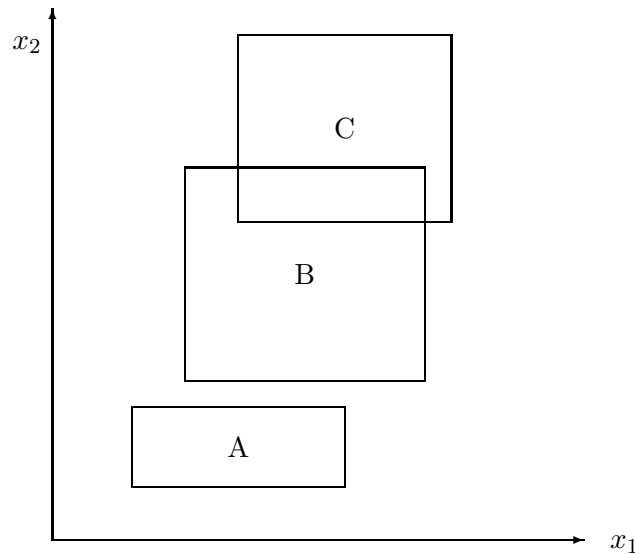


Figure II.10.: Three two-dimensional policies

The first step is to create the matrix and fill the first row. After the first step the matrix looks like this:

$$\begin{bmatrix} A & B & C \\ \{\} & \{\} & \{\} \\ \{\} & \{\} & \{\} \end{bmatrix}$$

After executing the *Conflict Detection Algorithm* for the first dimension ( $x_1$ ) the matrix looks like this:

$$\begin{bmatrix} A & B & C \\ \{B, C\} & \{C\} & \{\} \\ \{\} & \{\} & \{\} \end{bmatrix}$$

After executing the *Conflict Detection Algorithm* for the second and last dimension ( $x_2$ ) the matrix looks like this:

$$\begin{bmatrix} A & B & C \\ \{B, C\} & \{C\} & \{\} \\ \{\} & \{C\} & \{\} \end{bmatrix}$$

The only intersection that is left over is between  $B$  and  $C$ . From this results that the one and only conflict within these three policies is between policies  $B$  and  $C$ .

**Optimization** This data structure can be optimized to reduce memory usage by only creating a  $n \times 3$  matrix where the first row still contains the policies in order of the first attribute. The second and third row contain alternately the intersections of the dimension under inspection and the intersections of the preceding dimension.

#### II.4.5.2. Global Policy List

The policies that must be checked for conflicts must be held in a global list. We call this list *Global Policy List*. This list has a specialty because every policy entry has an additional boolean flag. This boolean flag indicates if the corresponding policy still must be considered ( $true \rightarrow \bullet$ ) or not ( $false \rightarrow \circ$ ). A policy must no longer be considered if it was pruned. The *Global Policy List* may look like:

$$\begin{pmatrix} A & B & C & D & E & F \\ \bullet & \bullet & \circ & \circ & \bullet & \bullet \end{pmatrix}.$$

The advantages of this *Global Policy List* is that it accelerate the intersection detection algorithms in the following way

- **Plane Sweep Algorithm:** The list of policies that must be sorted could get shorter with each dimension. The sorting is then faster and there are fewer intersection possibilities and that accelerates the detection of intersections.
- **AABB - AABB Intersection Algorithm:** The lists of policies that must be compared to each other could get shorter with each dimension. Fewer comparisons are needed and that accelerates the detection of intersections.
- **Regex Intersection Algorithm:** The list of policies that must be compared to each other could get shorter with each dimension. Fewer comparisons are needed and that accelerates the detection of intersections.

#### II.4.6. Runtime and Application Analysis

This section covers two main and important topics regarding the conflict detection algorithms described above

- **Runtime Analysis:** Analyzes the runtime of the algorithms described above.
- **Application Analysis:** Analyzes when which of the above described algorithms applies best for a given problem statement.

For the analyses we consider the following parameters

- $n$  denotes the number of policies
- $d$  denotes the number of dimensions (i.e. attributes)
- $p$  denotes the number of permit policies

#### II.4.6.1. Runtime Analysis

**Plane Sweep Algorithm** The *Plane Sweep Algorithm* consists of three phases. The first phase, *Sorting*, has a worst case runtime of

$$O(n * \log(n)).$$

The second and third phases, *Sweep the Plane* and *Report intersections*, run simultaneously and have a worst case runtime of

$$O(n).$$

Due to the fact that these algorithms run dimension by dimension both parts run  $d$  times. Therefore the first phase has a worst case runtime of

$$O(d * n * \log(n))$$

and the second and third phases have a worst case runtime of

$$O(d * n).$$

Summing up the *Plane Sweep Algorithm* has a worst case runtime of

$$O(d * n * \log(n)) + O(d * n) \subseteq O(d * n * \log(n)).$$

**AABB - AABB Intersection Algorithm** The *AABB - AABB Intersection Algorithm* consists of three phases. The first phase, *Create bounding sets*, has a worst case runtime of

$$O(n).$$

The second and third phase, *check for intersections of AABBs* and *report intersections*, run simultaneously and have a worst case runtime of

$$O(p * (n - p)).$$

This worst case behavior comes from the comparison of the two sets  $L$  and  $U$  element by element.

Due to the fact that these algorithms run dimension by dimension both parts run  $d$  times. Therefore the first phase has a worst case runtime of

$$O(d * n).$$

and the second and third phases have a worst case runtime of

$$O(d * p * (n - p)).$$

Summing up the *Plane Sweep Algorithm* has a worst case runtime of

$$O(d * n) + O(d * p * (n - p)) \subseteq O(d * p * (n - p)).$$

**Regex Intersection Algorithm** The *Regex Intersection Algorithm* consists of five phases. The first phase, *Transformation of the regular expressions into an  $\epsilon$ -free NFA*, has a worst case runtime of

$$O(n * l * \log^2(l))$$

where  $l$  is the number of symbols from the alphabet  $\Sigma$  in the regular expressions. Additionally the number of states in an NFA is also  $l$ . Hagenah and Muscholl [7] goes into further details on this topic.

The second phase, *Transformation of the  $\epsilon$ -free NFA into a DFA*, has a worst case runtime of

$$O(n * 2^l).$$

Due to the fact that the number of states in a DFA constructed from an NFA is  $2^l$  in maximum and the number of transitions is  $2^l * l$  in maximum, discussed by Hopcroft et al. [10]. The third phase, *Minimize the DFA*, has a worst case runtime of

$$O(n * 2^l * \log(2^l)).$$

Due to the fact that  $2^l$  is the number of states in the DFA. Hopcroft [9] goes into further details on this topic. The fourth phase, *Cartesian Product of all pairwise minimized DFA*, has a worst case runtime of

$$O\left(\binom{n}{2} * 2 * 2^{2*l} * l\right).$$

Due to the fact that  $2^l * l$  is the maximum number of possible transitions,  $2^l$  is the maximum number of possible states. For every combination of  $\binom{n}{2}$  automata every state of one automaton has to be connected to every state of the other plus all transitions have to be added to this new state. This has to be done vice versa. Han and Salomaa [8] goes into further details on this topic.

The fifth phase, *Depth-First Search*, has a worst case runtime of

$$O\left(\binom{n}{2} * 2 * 2^l * l + 2 * 2^l\right).$$

The basic formula for DFS is  $O(|V| + |E|)$  ( $V$  = Vertices and  $E$  = Edges). Here  $|V| = 2 * 2^l * l$  and  $|E| = 2 * 2^l$ . Knuth [14] goes into further details on this topic.

This algorithm runs also dimension by dimension. Therefore the final worst case runtimes are

- *Transformation of the regular expressions into an  $\epsilon$ -free NFA*:  $O(d * n * l * \log^2(l))$
- *Transformation of the  $\epsilon$ -free NFA into a DFA*:  $O(d * n * 2^l)$
- *Minimize the DFA*:  $O(d * n * 2^l * \log(2^l))$
- *Cartesian Product Construction of all pairwise minimized DFA*:  $O(d * \binom{n}{2} * 2 * 2^{2*l} * l)$
- *Depth-First Search*:  $O(d * \binom{n}{2} * 2 * 2^l * l + 2 * 2^l)$

### II.4.6.2. Application Analysis

The *Plane Sweep Algorithm* and the *AABB - AABB Intersection Algorithm* have fairly different runtimes (see Section II.4.6.1). The problem statement reveals which algorithm fits best for a certain problem. This section shall give an overview about the application area of those algorithms. The *Regex Intersection Algorithm* is noncompetitive because in case a dimension (i.e. attribute) has at least one *\*-match* comparison function it is required to use this algorithm.

The *Regex Intersection Algorithm* has a very bad worst case behavior. Due to this it is highly recommended to not use regular expressions at all. If it is indispensable to use regular expressions it is important to be careful to only use “simple” ones. “Simple” here means regular expressions that are short and do not have a lot of operations in it. Thereby the created automata are small and do not have a lot of transitions. This means they are manageable in a sensible amount of time.

The *Plane Sweep Algorithm* and the *AABB - AABB Intersection Algorithm* have quite different runtimes and have different applications as well. The *Plane Sweep Algorithm* is more efficient in a static environment and the *AABB - AABB Intersection Algorithm* is more efficient in a dynamic environment. A static environment means that the policies do not change very often and if they change then rather in a larger scale. A dynamic environment on the opposite has continuously a lot of rather small changes.

The *Plane Sweep Algorithm* is faster than the *AABB - AABB Intersection Algorithm* except for the case where the ratio between *Permit / Deny* policies is smaller than 1% for a large number of policies (> 100) and 5% to 20% for a small number of policies (< 100). Stepping one step back and have a look at those numbers it is easy to see that the *Plane Sweep Algorithm* is almost always the better choice. Having a ratio of less than 1% is fairly unrealistic and the cases where the ratio becomes beneficial (policies < 100) are negligible if considering the time spent within the algorithms in absolute numbers<sup>6</sup>.

Nevertheless the dynamic case described above is the case where the *AABB - AABB Intersection Algorithm* can go strong. The initial check for conflicts within the policies is slower than with the *Plane Sweep Algorithm*. But when keeping all expressions (pairwise policy comparisons) the *AABB - AABB Intersection Algorithm* runs in  $O(n)$  or  $O(d * n)$ , respectively, in all upcoming comparisons. This is faster than the  $O(n * \log(n))$  or  $O(d * n * \log(n))$ , respectively, of the *Plane Sweep Algorithm*.

### II.4.7. Runtime Optimization

Depending on the case there are possibilities to optimize the runtime of the algorithms. These optimizations require certain knowledge about the domain of application. If this pre-knowledge is available the algorithm could run much faster. The two possibilities, *Clustering* and *Attribute Ordering*, are described next.

#### II.4.7.1. Clustering

The policies can be divided into smaller sets that are guaranteed to be disjunct. The division must be made with the help of an attribute that is guaranteed to be not *multi-valued*. This means that the values of the attribute must not occur at the same time [12]. If such an

<sup>6</sup>It is negligible if the algorithm takes 2 seconds or 2.1 seconds

attribute is given these subsets  $S_i$  of the entire set of policies  $P$  ( $\forall_i S_i \subseteq P$ ) are handled independently by the algorithm due to the fact that a conflict among the different sets  $S_i$  cannot occur. This means that the number of comparison operations decreases and this accelerates the whole algorithm. Further the conflict detection algorithm for those sets  $S_i$  could run concurrently that could increase the runtime even more.

#### II.4.7.2. Attribute Ordering

A related approach would be to define the order of the attributes (i.e. the order of the dimensions). The advantages would be that domain knowledge could be used to accelerate the algorithm. The idea is to bring attributes that have a few or almost no intersections to the front (means that those are checked first for intersections). With this approach a lot of policies could be pruned out already at the beginning. Attributes that are known to not have any intersections could also be handled with the *Clustering* approach described above.

## III. Conflict Resolution

### III.1. Directed Acyclic Graph of Resolutions

The *Conflict Detection Algorithm*, see Section II.4, results in a set of pairwise conflicts. It is up to an administrator to resolve those conflicts in a reasonable way. When the conflicts are resolved a directed graph can be created when considering the policies as nodes and the resolutions as directed edges.

The resolution as directed edge means that if a policy  $B$  is overruled by a policy  $A$  (i.e.  $A$  precedes  $B$ ) we write  $A \rightarrow B$ . To ensure that the resolutions are consistent and reasonable the created graph must be a *directed acyclic graph (dag)*; i.e. cycles are not allowed.

A cycle denotes a desired conflict resolution that cannot be fulfilled. We illustrate this with a small example. Consider the policies in Figure III.1. The red-green-hatched areas

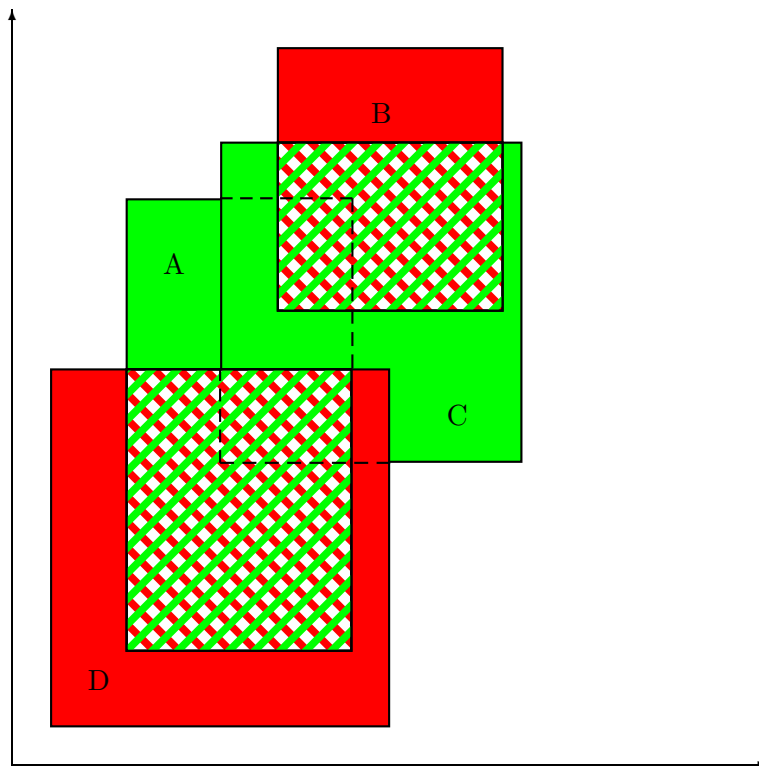


Figure III.1.: Overlapping Policies

denote conflicting areas (green equals *Permit* and red equals *Deny*). For such a set of policies the *Conflict Detection Algorithm* is going to return the conflicts

- $A - B$
- $A - D$
- $B - C$
- $C - D$

The overlap of A - C is not a conflict due to the fact that those have the same effect (*Permit*). If the administrator is going to resolve the conflicts like (a graphical illustration is in Figure III.1)

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow D$
- $D \rightarrow A$

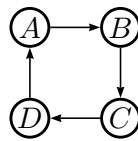


Figure III.2.: Stacking of slates not possible

results in a cycle that cannot be resolved. To illustrate this imagine the policies of Figure III.1 as slates and assume that always the most upper slate is the policy from which the effect is taken. When trying to stack those slates with the resolution from above it is not possible because the lowest slate then must also be the most upper slate. If the resolution would be (a graphical illustration is in Figure III.1)

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow D$
- $A \rightarrow D$

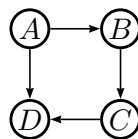


Figure III.3.: Stacking of slates possible

it is easily possible to stack the slates. This means it is also possible to resolve the conflicts. Summing up the slates are only stackable if the resolutions form a *directed acyclic graph*.

We define that a *starting node* is a node that is not preceded by another. This means it has only edges to other nodes but not from other nodes.

### III.1.1. Cycle Detection

It is necessary to detect cycles upfront. This is achieved by creating a graph from the resolutions the administrator has defined. In this graph the policies are the nodes and the resolutions are the edges.

To detect cycles in a directed graph we use *Tarjan's Algorithm* [19] to detect *Strongly Connected Components* in a graph.

A *Strongly Connected Component* denotes always a cycle within the graph. This cycle must be broken up by the administrator. It is not possible to automatically break up the cycles.

## III.2. Resolution Strategies

Hereafter we describe two strategies for resolving detected conflicts among policies. The result is a policy tree that is conflict free as the administrator intended it.

### III.2.1. Cutting Planes Algorithm

This approach of resolving conflicts changes the *Targets* of the policies in that way that only one policy is applicable at the same time. The name, *Cutting Planes*, is due to the geometric visualization of the policies where changing the *Targets* means cutting out a plane from the policy that is overruled by another. See Figure III.4 for a graphic illustration ( $P2$  precedes  $P1$ ). Here Figure III.4(a) shows the starting position: Two policies,  $P1$  and  $P2$ ,

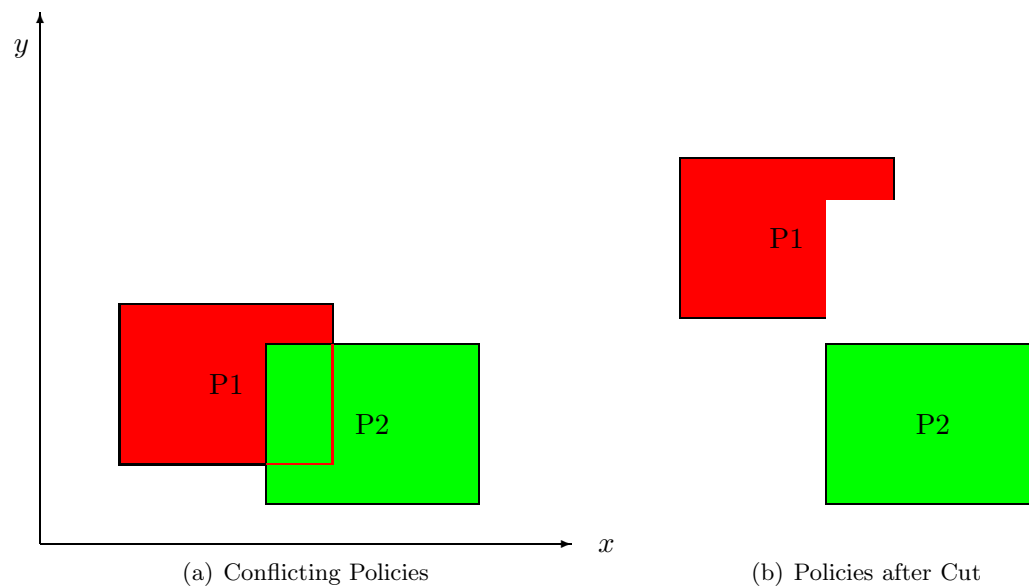


Figure III.4.: Cutting Planes Example

are conflicting. This conflict is resolved with  $P2 \rightarrow P1$ . The *Cutting Planes Algorithm* then cuts the conflicting plane from the overruled policy (here  $P1$ ), see Figure III.4(b). After completion the policies do no longer conflict.

---

#### Algorithm III.1 CuttingPlanes(dag)

---

- 1: //dag is the directed acyclic graph that represents the conflict resolutions
  - 2: **for all** root  $\in$  dag.rootNodes **do**
  - 3:   CuttingPlanes(root)
  - 4: **end for**
  - 5: createNewTargets(dag)
  - 6: **return** dag.policies
- 

After the cutting of the planes the policies can be combined under any combining algorithm, specified in the XACML 2.0 specification [2]. The order does not matter because only one policy is applicable at the same time.

We assume that a request does only have single-valued attributes [12]. This means that an attribute cannot have more than one value at the same time. If attributes exist that are multi-valued they must be handled in a special way, this is not handled within this thesis.

This resolution strategy can also be expressed in *fia terms*, described in Section I.6.1. In *fia terms* the precedence can be expressed with the *subtraction* operation. This means the precedence of  $P2$  over  $P1$  can be expressed as  $P2 \triangleright P1$ .

### III.2.1.1. Algorithm

A pseudo code description of the *Cutting Planes Algorithm* can be seen in Algorithm III.1 on page III.1. The algorithm depends on a *directed acyclic graph* described in Section III.1. The algorithm starts at every starting node of the *directed acyclic graph* independently. From each starting node the algorithm performs a *depth-first-search* through the graph. Every visited node is *cut* from its children. This means that the intersecting part is cut from the children. See Algorithm III.2. This means every preceding policy is cut from the overruled policy.

---

#### Algorithm III.2 CuttingPlanes(node)

---

```

1: //node is a policy from a directed acyclic graph
2: stack = node.children
3: while not stack.empty() do
4:   child = stack.pop()
5:   cut(node, child)
6:   CuttingPlanes(child)
7: end while

```

---

The *cut* operation is responsible of cutting the preceding policy from all its conflicting policies. The *cut* operation changes the *Target* of the policy that does not precede in that way that it is no longer applicable in the conflicting area (see Figure III.4(b) on page 55). To reach this goal the rectangle representing the *Target* is split into small rectangles. One of these rectangles is the conflicting area, this *conflicting rectangle* is then removed (see Figure III.5(b) on page 58). The remaining rectangles (here  $R_1$  and  $R_2$ ) represent the new conflict free *Target*. If further policies conflict, the algorithm is applied on the non-conflicting rectangles and not on the whole *Target* again. This is necessary to properly handle cases where more than two policies overlap in the same area.

The *cut* operation is described in pseudo code in Algorithm III.3 on page 57. The different operations within the *cut* operation are described hereafter:

- **project**: This operation projects the two nodes, i.e. policies, onto the given dimension.
- **createRange**: This operation creates a new range in the given dimension between the two given points.
- **expandArea**: This operation expands the conflicting area, that may span over multiple dimensions, with the range in the given dimension.

---

**Algorithm III.3** cut(node,childNode)

---

```
1: //node is the preceding policy.
2: //childNode is the overruled policy. This means the policy from which the conflicting
   area is cut out.
3: conflictingAreaUntilNow = {}
4: targetRectangles = {}
5: targetRectangles = childNode.targetRectangles
6: for all targetRectangle ∈ targetRectangles do
7:   for all dimension ∈ dimensions do
8:     previousPrecedingNodeWasActive = false
9:     previousOverruledNodeWasActive = false
10:    previousPoint = null
11:    conflictingAreaCurrent = null
12:    projectionPoints = project(dimension, node, targetRectangle)
13:    for all point ∈ projectionPoints do
14:      if previousPrecedingNodeWasActive and previousOverruledNodeWasActive
        then
15:        conflictingAreaCurrent = expandArea(conflictingAreaUntilNow,
16:        createRange(dimension,previousPoint,point))
17:      end if
18:      if not previousPrecedingNodeWasActive and previousOverruledNodeWasActive
        then
19:        targetRectangles.add(completeWithUpcomingDimensions(
20:        expandArea(conflictingAreaUntilNow,createRange(dimension,previousPoint,point))))
21:      end if
22:      if point belongs to node then
23:        previousPrecedingNodeWasActive = not previousPrecedingNodeWasActive
24:      end if
25:      if point belongs to childNode then
26:        previousOverruledNodeWasActive = not previousOverruledNodeWasActive
27:      end if
28:      previousPoint = point
29:    end for
30:    conflictingAreaUntilNow = conflictingAreaCurrent
31:  end for
32:  if targetRectangle equals conflictingAreaUntilNow then
33:    targetRectangles.remove(targetRectangle)
34:  end if
35: end for
36: childNode.setNewTargets(targetRectangles)
```

---

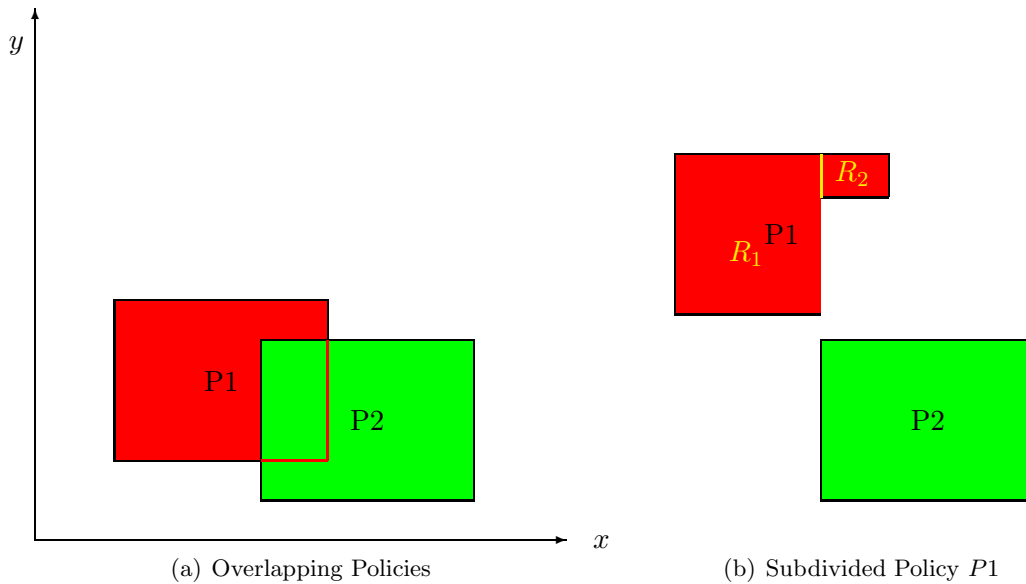


Figure III.5.: Cutting Planes Algorithm

- **add:** This operation (belonging to the list of target rectangles) adds a new target rectangle to the list. If the new rectangle is a sub-rectangle of a rectangle that is already in the list, this super-rectangle must be deleted first.
- **completeWithUpcomingDimensions:** This operation completes the given area (that may span over multiple dimensions) with all upcoming dimensions. The upcoming dimensions shall always cover the whole range the child node (i.e. overruled policy) covers.
- **setNewTargets:** This operation (belonging to the policies) sets a new target that consists of the given rectangles.

**Hidden policies** In case a policy  $A$  is fully enclosed by another policy  $B$  and policy  $B$  precedes  $A$ , then  $A$  can be removed because it is unreachable.

### III.2.1.2. Examples

Hereafter we show some examples of conflicting policies and their conflict resolution with the *Cutting Planes Algorithm*. The rectangles created by the *cut* operation have a yellow border. The new *Target* is the combination of all yellow rectangles.

The first example is an extended example that shows some intermediate output of the algorithm as well.

**Example 1** Consider the following two policies:

- $P1 : \text{if}((1 \leq x \leq 3) \text{ AND } (2 \leq y \leq 4)) \rightarrow \text{Deny}$
- $P2 : \text{if}((2 \leq x \leq 4) \text{ AND } (1 \leq y \leq 3)) \rightarrow \text{Permit}$

where  $P2 \rightarrow P1$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.6.

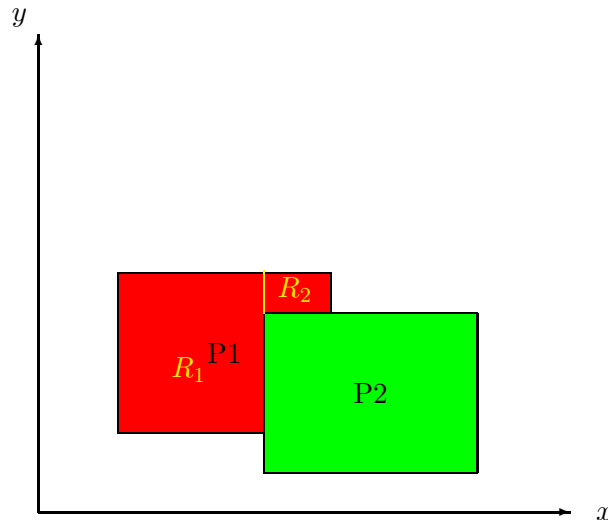


Figure III.6.: Example 1

The conflicting part is cut out and the rest of  $P1$  is split into multiple rectangles (here  $R_1$  and  $R_2$ ). Next we will show some important steps of the algorithm (only of the *cut* operation) in detail (hint: There is only one *targetRectangle*): The following steps only show changes to particular variables within the algorithm.

- Iteration over the first dimension  $x$ .
  - $previousPrecedingNodeWasActive = false$
  - $previousOverruledNodeWasActive = false$
  - $previousPoint = null$
  - $conflictingAreaCurrent = null$
  - 1.  $previousPrecedingNodeWasActive = false$   
 $previousOverruledNodeWasActive = true$
  - 2.  $targetRectangles += (\{1, 2\}_x, \{2, 4\}_y)$   
 $previousPrecedingNodeWasActive = true$   
 $previousOverruledNodeWasActive = true$
  - 3.  $conflictingAreaCurrent = \{2, 3\}_x$   
 $previousPrecedingNodeWasActive = true$   
 $previousOverruledNodeWasActive = false$
  - 4. *nothing to do...*
  - 5.  $conflictingAreaUntilNow = conflictingAreaCurrent = \{2, 3\}_x$
- Iteration over the second dimension  $y$ .
  - $previousPrecedingNodeWasActive = false$
  - $previousOverruledNodeWasActive = false$
  - $previousPoint = null$
  - $conflictingAreaCurrent = null$

1.  $previousPrecedingNodeWasActive = true$   
 $previousOverruledNodeWasActive = false$
  2.  $previousPrecedingNodeWasActive = true$   
 $previousOverruledNodeWasActive = true$
  3.  $conflictingAreaCurrent = \{2, 3\}_y$   
 $previousPrecedingNodeWasActive = false$   
 $previousOverruledNodeWasActive = true$
  4.  $targetRectangles += conflictingAreaUntilNow + \{3, 4\}_y = (\{2, 3\}_x, \{3, 4\}_y)$
  5.  $conflictingAreaUntilNow = conflictingAreaCurrent = \{2, 3\}_y$
- $childNode.setNewTargets((\{1, 2\}_x, \{2, 4\}_y), (\{2, 3\}_x, \{3, 4\}_y))$

**Example 2** Consider the following two policies:

- $P1 : if((1 \leq x \leq 3) AND (2 \leq y \leq 3)) \rightarrow Deny$
- $P2 : if((2 \leq x \leq 4) AND (1 \leq y \leq 4)) \rightarrow Permit$

where  $P1$  shall precede  $P2$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.7.

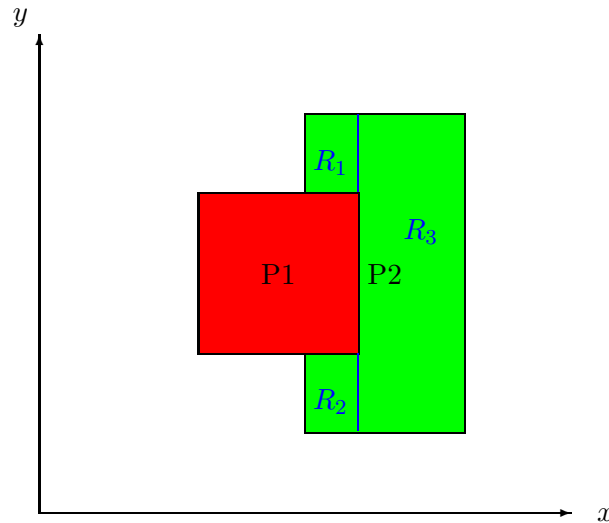


Figure III.7.: Example 2

**Example 3** Consider the following two policies:

- $P1 : if((1 \leq x \leq 3) AND (1 \leq y \leq 2)) \rightarrow Deny$
- $P2 : if((2 \leq x \leq 4) AND (1 \leq y \leq 2)) \rightarrow Permit$

where  $P2$  shall precede  $P1$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.8.

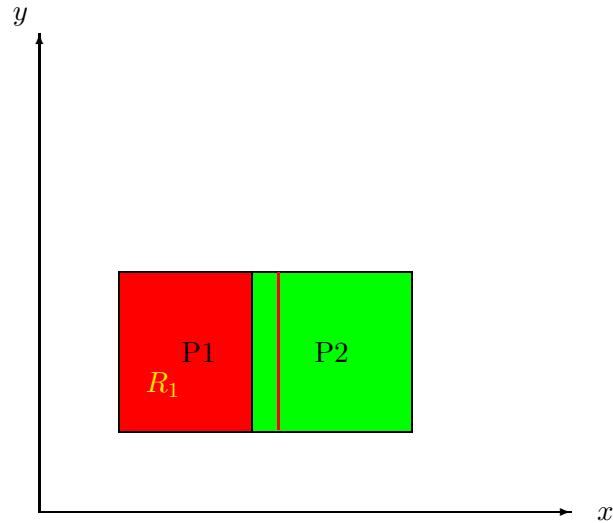


Figure III.8.: Example 3

This example is a special case where the policies have equal ranges in a dimension (here  $y$ ). Here the resulting new target rectangle is equal to the visible red part (see Figure III.8).

**Example 4** Consider the following two policies:

- $P1 : \text{if}((1 \leq x \leq 3) \text{ AND } (1 \leq y \leq 4)) \rightarrow \text{Deny}$
- $P2 : \text{if}((2 \leq x \leq 4) \text{ AND } (2 \leq y \leq 3)) \rightarrow \text{Permit}$

where  $P1$  shall precede  $P2$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.9.

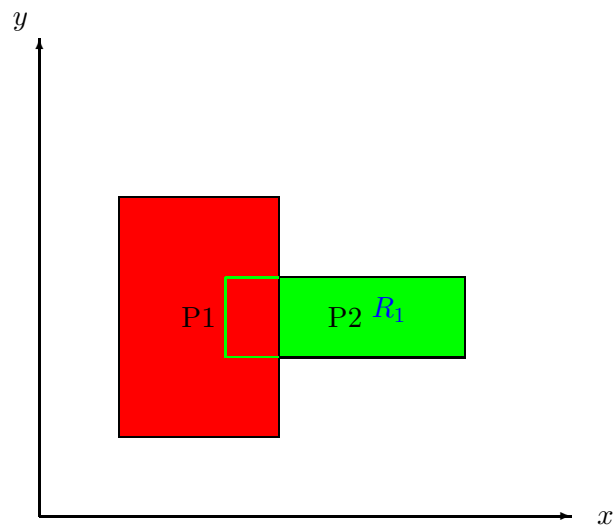


Figure III.9.: Example 4

Here the resulting new target rectangle is equal to the visible green part (see Figure III.9).

**Example 5** Consider the following two policies:

- $P1 : \text{if}((1 \leq x \leq 3) \text{ AND } (2 \leq y \leq 4)) \rightarrow \text{Deny}$
- $P2 : \text{if}((2 \leq x \leq 4) \text{ AND } (1 \leq y \leq 3)) \rightarrow \text{Permit}$

where  $P1$  shall precede  $P2$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.10.

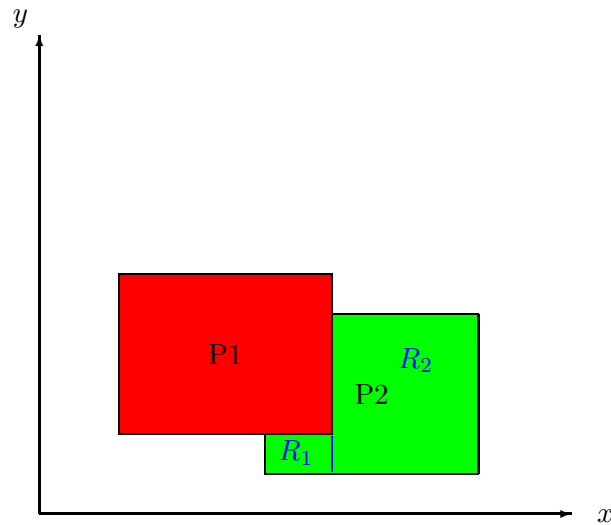


Figure III.10.: Example 5

**Example 6** Consider the following two policies:

- $P1 : \text{if}((1 \leq x \leq 3) \text{ AND } (2 \leq y \leq 4)) \rightarrow \text{Permit}$
- $P2 : \text{if}((2 \leq x \leq 4) \text{ AND } (1 \leq y \leq 3)) \rightarrow \text{Permit}$

where  $P2$  shall precede  $P1$ .

The application of the *Cutting Planes Algorithm* provides the result that can be seen in Figure III.11 on page III.11.

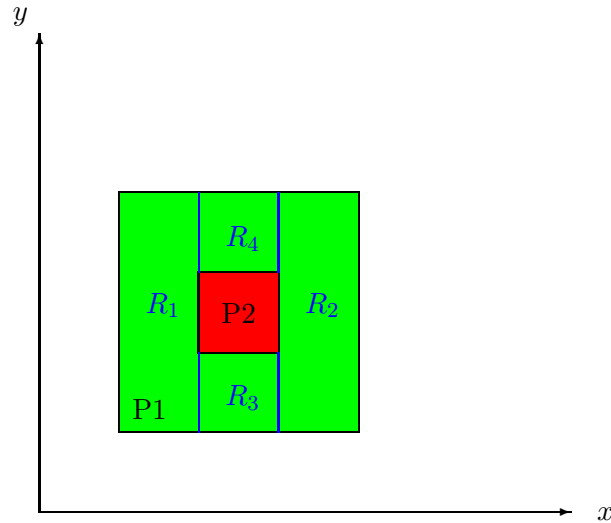


Figure III.11.: Example 6

A special case is if  $P1$  would precede  $P2$ . This would mean that  $P2$  is fully covered by  $P1$  and will never be applicable. The *Cutting Planes Algorithm* would then set an empty list of *targetRectangles* on the policy  $P2$ . If a policy has an empty list of *targetRectangles* it can be removed.

**Example 7** This example shows how the *Cutting Planes Algorithm* works if more than two (here three) policies conflict.

Consider the following three policies:

- $P1 : \text{if}((1 \leq x \leq 4) \text{ AND } (3 \leq y \leq 5)) \rightarrow \text{Permit}$
- $P2 : \text{if}((3 \leq x \leq 6) \text{ AND } (2 \leq y \leq 6)) \rightarrow \text{Deny}$
- $P3 : \text{if}((2 \leq x \leq 5) \text{ AND } (1 \leq y \leq 4)) \rightarrow \text{Permit}$

where  $P1$  and  $P3$  both shall precede  $P2$ .

First the conflict between  $P1$  and  $P2$  is resolved. See Figure III.12.

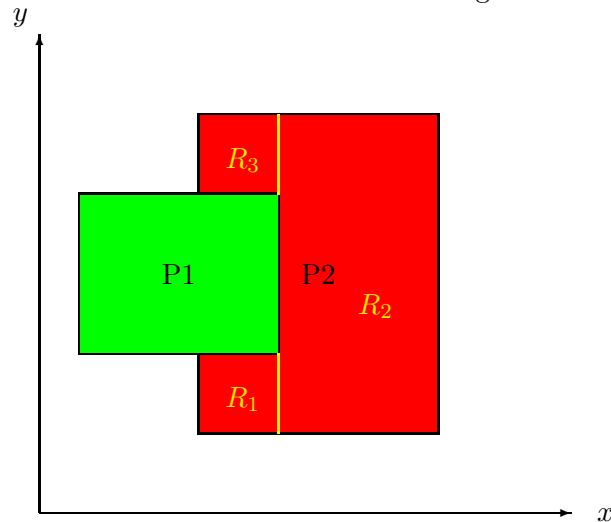


Figure III.12.: Example 7,  $P1 \rightarrow P2$

The next step after the resolution of  $P1 \rightarrow P2$  is the resolution of  $P3 \rightarrow P2$ . The *Target* of  $P2$  consists now of three *targetTectangles*. The situation looks as in Figure III.13.

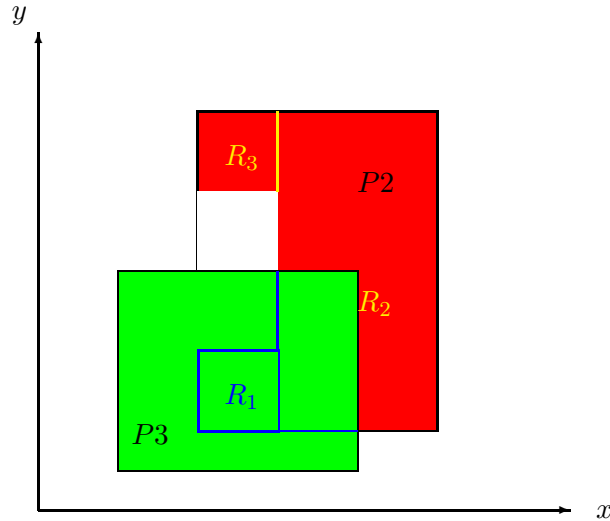


Figure III.13.: Example 7,  $P3 \rightarrow P2$

The resolution of  $P3 \rightarrow P2$  is split into three separate cases:

- $P3 \rightarrow R_1$ , see Figure III.15(a) on page 65
- $P3 \rightarrow R_2$ , see Figure III.15(b) on page 65
- $P3 \rightarrow R_3$ , see Figure III.15(c) on page 65

After the completion of the algorithm the resulting policy  $P2$  looks like Figure III.14.

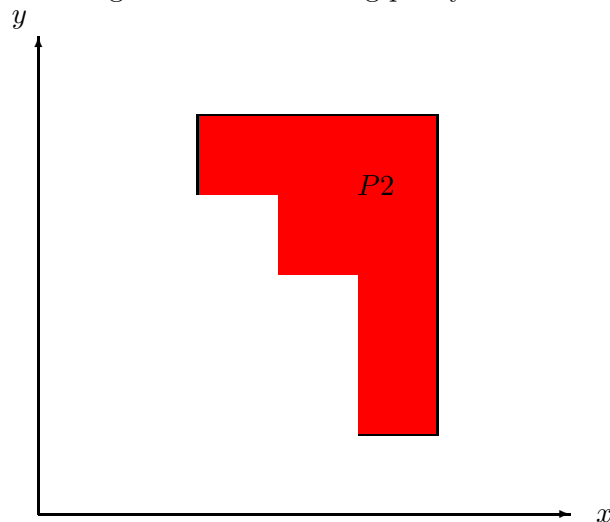


Figure III.14.: Example 7,  $P2$  resolved

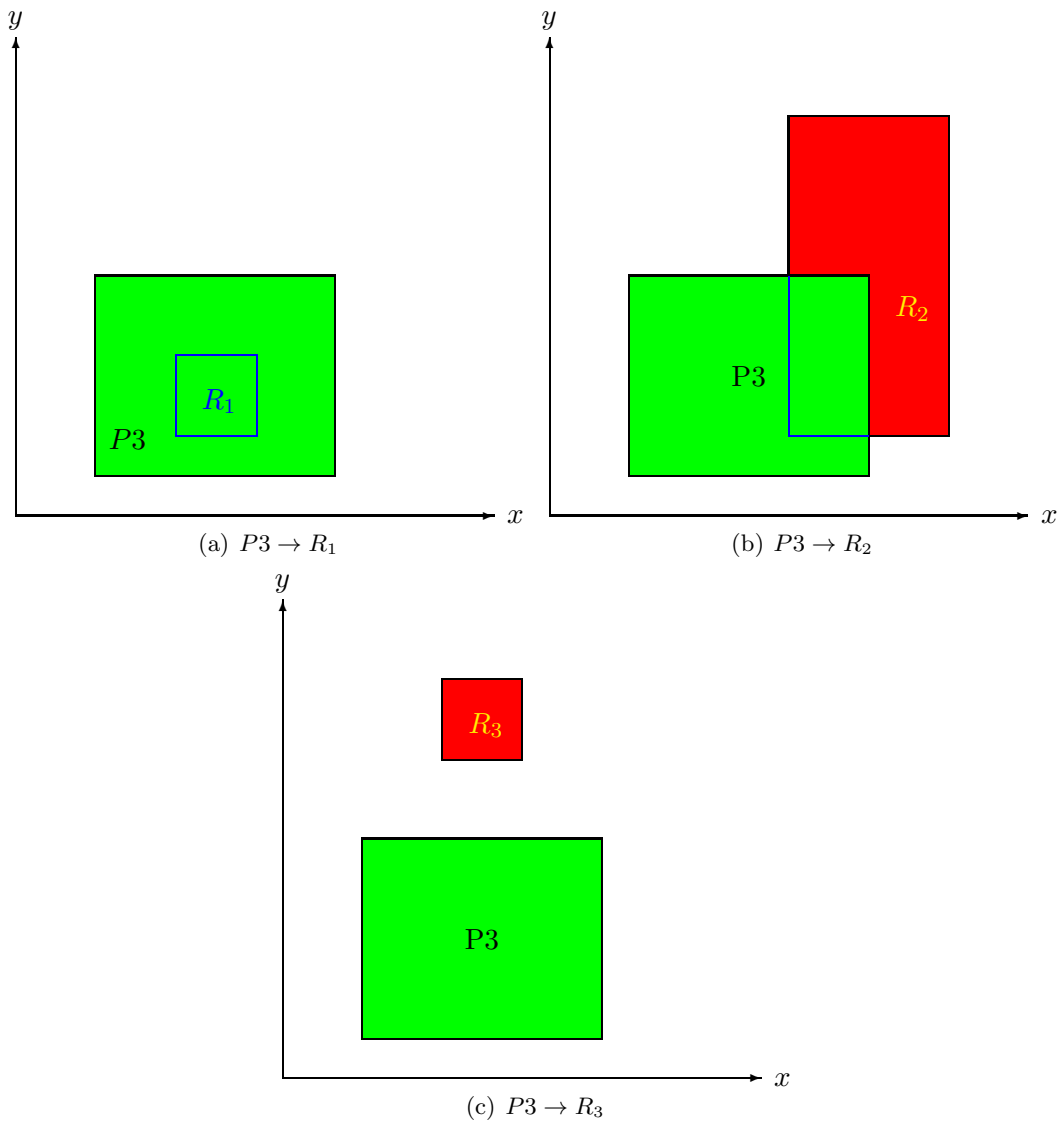


Figure III.15.: Example 7,  $P3 \rightarrow P2$  (splitted)

### III.2.2. Precedence Stringing

This approach of resolving conflicts makes use of the *first-applicable* semantics, specified in the XACML 2.0 specification [2]. *first-applicable* is specified as all policies are stringed together and the evaluation runs through the policies in the order they were stringed. The decision of the first applicable policy is then returned and the evaluation finishes.

The algorithm to string the policies together can be seen in pseudo code in Algorithm III.4.

The main idea behind the algorithm is to process every reachable node in the *dag* with either a *Depth-First-Search* or a *Breadth-First-Search* starting at the starting nodes. The nodes then are numbered with their distance to the starting node. If a node is processed more than once always the higher distance is taken as the distance of the node to the starting node, see pseudo code in Algorithm III.5. The reason to take the higher distance is due to the fact that a higher distance does not change the order of a path through the *dag* but allows other paths to integrate themselves into the order. In the next step the nodes are sorted according to their distance. The sorted nodes then represent the order in which the policies shall be stringed together for a *first-applicable* evaluation. The

---

#### Algorithm III.4 PrecedenceStringing(dag)

---

```

1: //dag is the directed acyclic graph that represents the conflict resolutions
2: for all root ∈ dag.startingNodes do
3:   //Initially the starting nodes have a distance of 0
4:   PrecedenceStringingDistanceEvaluation(root)
5: end for
6: sortedList = sort(dag.policies) //Sorts the policies by distance (ascending)
7: return sortedList

```

---



---

#### Algorithm III.5 PrecedenceStringingDistanceEvaluation(node)

---

```

1: //node is a policy from a directed acyclic graph
2: stack = node.children
3: while not stack.empty() do
4:   child = stack.pop()
5:   if child.distance < node.distance then
6:     child.distance |= node.distance + 1
7:   end if
8:   PrecedenceStringingDistanceEvaluation(child)
9: end while

```

---

order of nodes that have the same distance does not matter. It can easily be seen that the algorithm's output is as expected because it is guaranteed that the preceding policies have a lower distance and always come prior to the overruled policies.

#### III.2.2.1. Example

An example of the *Precedence Stringing Algorithm* is shown in Table III.1 on page 67. The example follows Algorithm III.4. It is a simple example to show the idea behind the algorithm.

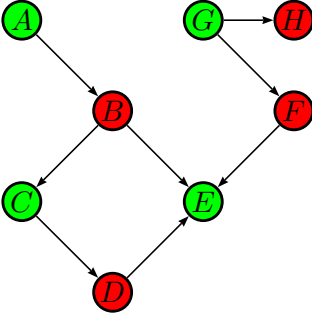
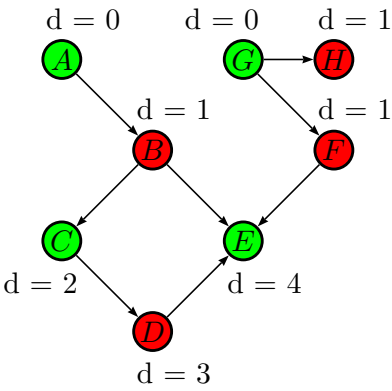

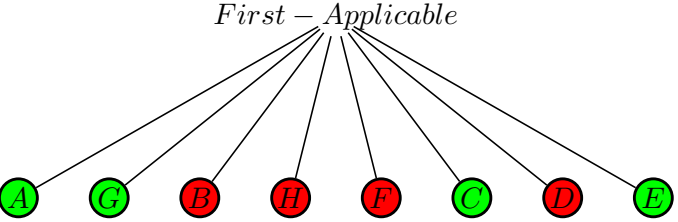
<p>The algorithm starts at the starting nodes (<math>A</math> and <math>G</math>) of the directed acyclic graph (<math>dag</math>) representing the resolution of the conflicts.</p>	
<p>The next step is to set the maximum distance <math>d</math> of all nodes.</p>	
<p>The next step is to sort the nodes regarding their distance.</p>	
<p>The last step is to combine the policies under the <i>first-applicable</i> combining algorithm.</p>	<p style="text-align: center;"><i>First - Applicable</i></p> 

Table III.1.: Precedence Stringing Example

The node  $E$  is reachable through three paths:

- $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$
- $A \rightarrow B \rightarrow E$
- $G \rightarrow F \rightarrow E$

As described the effective distance of a node must always be the maximum distance to a starting node. It does not matter which starting node. The only criteria is the maximum distance. Therefore node  $E$  gets here the distance 4 as it can be seen in Table III.1 on page 67.

### III.2.3. Analysis

#### III.2.3.1. Cutting Planes

The *Cutting Planes Algorithm* performs a *depth-first-search* through the nodes starting at each starting node. For each node the *cut* operations runs through all dimensions (i.e. attributes) to do the cutting. The overall runtime of the *Cutting Planes Algorithm* is:

$$O(r * n * c * d)$$

where  $r$  is the number of starting nodes,  $n$  the number of policies,  $c$  the number of conflicts and  $d$  the number of dimensions.  $n * c$  is due to the *depth-first-search*.

#### III.2.3.2. Precedence Stringing

The *Precedence Stringing Algorithm* performs a *depth-first-search* or a *breath-first-search* starting at each starting node. The overall runtime of the *Precedence Stringing Algorithm* is:

$$O(n * c)$$

$n$  is the number of policies and  $c$  the number of conflicts. It does not matter which search algorithm is used, both are equally regarding their runtime.

#### III.2.3.3. Comparison

Regarding the runtime and complexity of the algorithm the *Precedence Stringing Algorithm* is clearly the better choice to take. At first glance it seems that the big advantage of the *Cutting Planes Algorithm* is the characteristic that only one policy is applicable for a request. In conjunction with an indexed policy repository it would only be necessary to evaluate one single policy per request.

By taking a closer look this advantage is not really one in practice. The same indexing approach could also be applied on the policies returned by the *Precedence Stringing Algorithm*, the difference is that in this case the index may return more than one policy. It is questionable if this is a real drawback.

We would recommend to use the *Precedence Stringing Algorithm* by default and only use the *Cutting Planes Algorithm* in special applications.

But the *Cutting Planes Algorithm* demonstrates another very interesting thing about XACML policies in the  $n$ -dimensional space. The approach of cutting out planes can be used to create two figures in the  $n$ -dimensional space, a *Permit*- and a *Deny*-figure. These two figures then can be combined and split as a given scenario requires it. From this it follows that we are able to restructure XACML policies regarding the problem statement as we want.

### III.3. Default Decision

The *default decision* is a decision that is returned in case where no policy is applicable. A *default decision* can very easily be achieved. A new root node is introduced that contains two nodes. The first is the “old” root node of the policy tree that results from the conflict resolution and the second node is the default decision. This default decision node is a policy (that contains a *Rule*) that has no *Target* and no *Condition*. The only relevant thing is the *Effect* of the *Rule* that states the *default decision*.

The (new) root node must be a *PolicySet* that precedes the contrary decision than the *default decision*. E.g. if the *default decision* is *deny* than the root *PolicySet* must precede *permit*.

## IV. Conclusion

## IV.1. Achievements

In this thesis we have proposed three algorithms for *Conflict Detection* and two algorithms for *Conflict Resolution*.

The *Conflict Detection Algorithms* have three different application areas. The *Plane Sweep Algorithm* is intended to be used in a static environment where the policy structure does not change a lot after an initial deployment. The *AABB - AABB Intersection Algorithm* is intended to be used in a dynamic environment where the policy structure is continuously changing. This means that policies are added and removed from the deployed set all the time. The *Regex Intersection Algorithm* is intended to be used in cases where intersections between regular expressions must be detected.

The *Conflict Detection Algorithms* are generic. This means they can be used to detect any kind of conflicts. It only has to be defined what a conflict is and the algorithm can detect those. An example would be to detect *Obligation-Conflicts*. The *Conflict Resolution Algorithms* both follow a very different approach but the resulting evaluation time of the policies is almost equal. The *Cutting Planes Algorithm* cuts the intersecting part of two policies from the overruled policy. The result is that on each request only one policy is applicable at the same time. The *Precedence Stringing Algorithm* orders the policies according to their precedence and combines all of them under the *first-applicable* combining algorithm.

Due to the fact that the *Cutting Planes Algorithm* is much more complex than the *Precedence Stringing Algorithm* we recommend to use the latter for *Conflict Resolution*. The *Cutting Planes Algorithm* can, with a few small adaptations, be used as a tool for policy restructuring. It allows the cutting and recombination of policies into new combinations. With this algorithm an administrator would be able to optimize the policies for evaluation.

These algorithms as a whole enable an administrator to create and deploy policies that are free of any conflicts.

A demonstration of the *AABB - AABB Intersection Algorithm* and of the *Precedence Stringing Algorithm* can be downloaded from the *Research Section* on the HERAS<sup>AF</sup> Home-page<sup>1</sup>.

---

<sup>1</sup><http://www.herasaf.org>

## IV.2. Further Work

At the beginning we excluded the special cases of *policy references* and *multi-valued attributes* from this thesis. For a productive application these two cases must be worked out to obtain a full coverage of possible situations. After supporting these two cases the *Conflict Detection Algorithms* and the *Conflict Resolution Algorithms* can be used in every XACML-based policy authoring tool to be able to deploy a set of policies that is free of any conflicts as intended.

A possible solution for the handling of *multi-valued attributes* could be to introduce a data type that can represent *multi-valued attributes*. Another solution could be to exclude the *multi-valued attributes* from the conflict detection and only use them for conflict resolution. This must be further investigated.

Further we excluded the consideration of the *Condition* of an XACML *Rule*. Due to the fact that the *Rule* is a restriction to the XACML *Target* it might be that a conflict between two policies (based on their *Targets*) is in fact not a conflict because their *Conditions* restrict the area of applicability that much that the policies do no longer conflict. It is questionable if it makes sense to do a conflict detection on such a deep level but shall be further investigated.

## A. Examples

## A.1. FIA Examples

In Section I.6.1 we have described the *fine-grained integration algebra*. Hereafter we show some XACML example of this algebra. For all examples in this section we define the following vocabulary  $\Sigma$ :  $\Sigma = \{\text{name, hair}\}$ . We assume that  $R_\Sigma$  is:  $\{(name = "Max"), (name = "Sophie"), (hair = "brown"), (name = "Max", hair = "brown"), (name = "Sophie", hair = "brown")\}$ . We assume that a request always holds the attribute *name*. Further we define the following XACML policies:

Listing A.1: Policy 1

```
<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy1"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
  algorithm:permit-overrides">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
        -equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
          string">Max</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="name" />
        </SubjectMatch>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
        -equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
          string">brown</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="hair" />
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources />
    <Actions />
    <Environments />
  </Target>
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Permit">
  </Rule>
</Policy>
```

Listing A.2: Policy 2

```
<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy2"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-
  overrides">
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Deny">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
          function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
            string">Max</AttributeValue>
```

```

        <SubjectAttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#string" ←
          AttributeId="name" />
      </SubjectMatch>
    <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0←
      :function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#←
        string">brown</AttributeValue>
      <SubjectAttributeDesignator
        DataType="http://www.w3.org/2001/XMLSchema#string" ←
        AttributeId="hair" />
    </SubjectMatch>
  </Subject>
</Subjects>
<Resources />
<Actions />
<Environments />
</Target>
</Rule>
<Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule2"
  Effect="Permit">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0←
          :function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#←
            string">Sophie</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string" ←
            AttributeId="name" />
        </SubjectMatch>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0←
          :function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#←
            string">brown</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string" ←
            AttributeId="hair" />
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources />
    <Actions />
    <Environments />
  </Target>
</Rule>
</Policy>

```

The corresponding policies in FIA terms look the following:

$$\begin{aligned}
 P_1 &= \langle \{(name = "Max", hair = "brown")\}, \emptyset, R_\Sigma \setminus \{(name = "Max", hair = "brown")\} \rangle \\
 P_2 &= \langle \{(name = "Sophie", hair = "brown")\}, \{(name = "Max", hair = "brown")\}, R_\Sigma \setminus \\
 &\quad \{(name = "Sophie", hair = "brown"), (name = "Max", hair = "brown")\} \rangle
 \end{aligned}$$

### A.1.1. Addition

$$P_I = P_1 + P_2 = \langle \{ (name = \text{"Sophie"}, hair = \text{"brown"}), (name = \text{"Max"}, hair = \text{"brown"}) \}, \emptyset, R_\Sigma \setminus \{ (name = \text{"Sophie"}, hair = \text{"brown"}), (name = \text{"Max"}, hair = \text{"brown"}) \} \rangle$$

Listing A.3:  $P_1 + P_2$

```

<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy1"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
  algorithm:permit-overrides">
  <Target />
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">Max</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">brown</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="hair" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources />
      <Actions />
      <Environments />
    </Target>
  </Rule>
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule2"
    Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">Sophie</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">brown</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="hair" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources />
      <Actions />
    </Target>
  </Rule>
</Policy>

```

```

    <Environments />
  </Target>
</Rule>
</Policy>

```

### A.1.2. Intersection

$$P_I = P_1 \& P_2 = \langle \emptyset, \emptyset, R_\Sigma \rangle$$

This intersection result means that the policy can be removed because it is always NA.

### A.1.3. Negation

$$P_I = \neg P_1 = \langle \emptyset, \{(name = "Max", hair = "brown")\}, R_\Sigma \setminus \{(name = "Max", hair = "brown")\} \rangle$$

Listing A.4:  $\neg P_1$

```

<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy1"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Max</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="name" />
        </SubjectMatch>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">brown</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="hair" />
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources />
    <Actions />
    <Environments />
  </Target>
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Deny">
  </Rule>
</Policy>

```

### A.1.4. Domain Projection

Here we assume that  $hair \in A$  and  $name \in B$  and A and B denote two different domains.

$$P_I = \Pi_B(P_1) = \langle \{name\text{"Max"}\}, \emptyset, R_\Sigma \setminus \{name\text{"Max"}\} \rangle$$

Listing A.5:  $\Pi_B(P_1)$

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy1"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
  algorithm:permit-overrides">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
        -equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
          string">Max</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="name" />
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources />
    <Actions />
    <Environments />
  </Target>
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Permit">
  </Rule>
</Policy>
```

### A.1.5. Effect Projection

$$P_I = \Pi_Y(P_2) = \langle \{(name = \text{"Sophie"}, hair = \text{"brown"})\}, \emptyset, \emptyset \rangle$$

Listing A.6:  $\Pi_Y(P_2)$

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy2"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
  algorithm:deny-overrides">
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule2"
    Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
          :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
            string">Sophie</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
          :function:string-equal">
```

```

        <AttributeValue DataType=" http://www.w3.org/2001/XMLSchema#string">brown</AttributeValue>
        <SubjectAttributeDesignator
          DataType=" http://www.w3.org/2001/XMLSchema#string" ←
          AttributeId="hair" />
      </SubjectMatch>
    </Subject>
  </Subjects>
</Resources />
</Actions />
</Environments />
</Target>
</Rule>
</Policy>

```

### A.1.6. Subtraction

$$P_I = P_2 - P_1 = \{(name = "Sophie", hair = "brown")\}, \emptyset, \emptyset$$

Listing A.7:  $P_2 - P_1$

```

<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy2"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-←
  overrides">
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule2"
    Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:←
          :function:string-equal">
            <AttributeValue DataType=" http://www.w3.org/2001/XMLSchema#←
            string">Sophie</AttributeValue>
            <SubjectAttributeDesignator
              DataType=" http://www.w3.org/2001/XMLSchema#string" ←
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:←
          :function:string-equal">
            <AttributeValue DataType=" http://www.w3.org/2001/XMLSchema#←
            string">brown</AttributeValue>
            <SubjectAttributeDesignator
              DataType=" http://www.w3.org/2001/XMLSchema#string" ←
              AttributeId="hair" />
          </SubjectMatch>
        </Subject>
      </Subjects>
    </Resources />
    </Actions />
    </Environments />
  </Target>
</Rule>
</Policy>

```

## A.1.7. Precedence

$P_1 = P_1 \triangleright P_2 = \{(name = \text{"Sophie"}, hair = \text{"brown"}), (name = \text{"Max"}, hair = \text{"brown"})\}, \emptyset, \emptyset\}$

Listing A.8:  $P_1 \triangleright P_2$

```
<?xml version="1.0" encoding="UTF-8" ?>
<Policy PolicyId="urn:org:herasaf:xacml:examplePolicy:fia:policy:policy2"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
  <Rule RuleId="urn:org:herasaf:xacml:examplePolicy:fia:rule:rule1"
    Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">Max</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">brown</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="hair" />
          </SubjectMatch>
        </Subject>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">Sophie</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="name" />
          </SubjectMatch>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
            :function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
              string">brown</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="hair" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources />
      <Actions />
      <Environments />
    </Target>
  </Rule>
</Policy>
```

## Bibliography

- [1] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, 2002. ISSN 1094-9224. doi: <http://doi.acm.org/10.1145/504909.504910>.
- [2] XACML Technical Committee. eXtensible Access Control Markup Language (XACML) Version 2.0. Specification Errata 29. January 2008, OASIS, 2008.
- [3] David H. Crocker. RFC822 - standard for the format of arpa internet text messages, 1982.
- [4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry, Algorithms and Applications*. Springer, third edition edition, 1998.
- [5] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.
- [6] P. Faltstrom, P. Hoffman, and A. Costello. RFC3490 - internationalizing domain names in applications (idna), 2003.
- [7] Christian Hagenah and Anca Muscholl. Computing  $\epsilon$ -free nfa from regular expressions in  $O(n \log^2(n))$  time. Technical report, Institut für Informatik, Universität Stuttgart, 1998.
- [8] Yo-Sub Han and Kai Salomaa. State complexity of union and intersection of finite languages. *International Journal of Foundations of Computer Science*, 19(3):581–595, 2008.
- [9] John E. Hopcroft. An  $n \log n$  algorithm for minimizing states in a finite automaton. Technical report, Stanford University, 1971.
- [10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 3 edition, 2006.
- [11] R. Housley, W. Polk, W. Ford, and D. Solo. RFC3280 - internet X.509 public key infrastructure, 2002.
- [12] Florian Huonder. Analysis of access control policies. Master Student Research Project, 2008. University of Applied Sciences Rapperswil.
- [13] J. Klensin. RFC2821 - simple mail transfer protocol, 2001.
- [14] Donald Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, third edition, 1997.
- [15] Donald Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, second edition, 1998.

- [16] Stefan Oberholzer. Indexing of access control policies. Master Student Research Project, 2010. University of Applied Sciences Rapperswil.
- [17] Prathima Rao, Dan Lin, Elisa Bertino, Ninghui Li, and Jorge Lobo. An algebra for fine-grained integration of XACML policies. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 63–72, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-537-6. doi: <http://doi.acm.org/10.1145/1542207.1542218>.
- [18] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.
- [19] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [20] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/363347.363387>.
- [21] International Telecommunication Union. ITU-T X.690, 2002.
- [22] M. Wahl, S. Kille, and T. Howes. RFC2253 - lightweight directory access protocol (v3): UTF-8 string representation of distinguished names, 1997.