

Access Control Policies: Modeling and Validation

Luigi Logrippo

&

Mahdi Mankai

Université du Québec en Outaouais

Overview

- Introduction
- XACML overview
- A Logical Model of XACML
- Modeling with Alloy
- Access Control Verification and Validation
- Related Work
- Conclusion
- Future work

Introduction

- Access control policies languages
 - XACML
 - EPAL
 - PONDER
 - ...
- Possible inconsistencies within policies
- How to solve inconsistencies at execution time
 - Precedence rules
 - Priorities
- How to detect inconsistencies at design time
 - First-order logic
 - Model-checking tools

An example

Subject

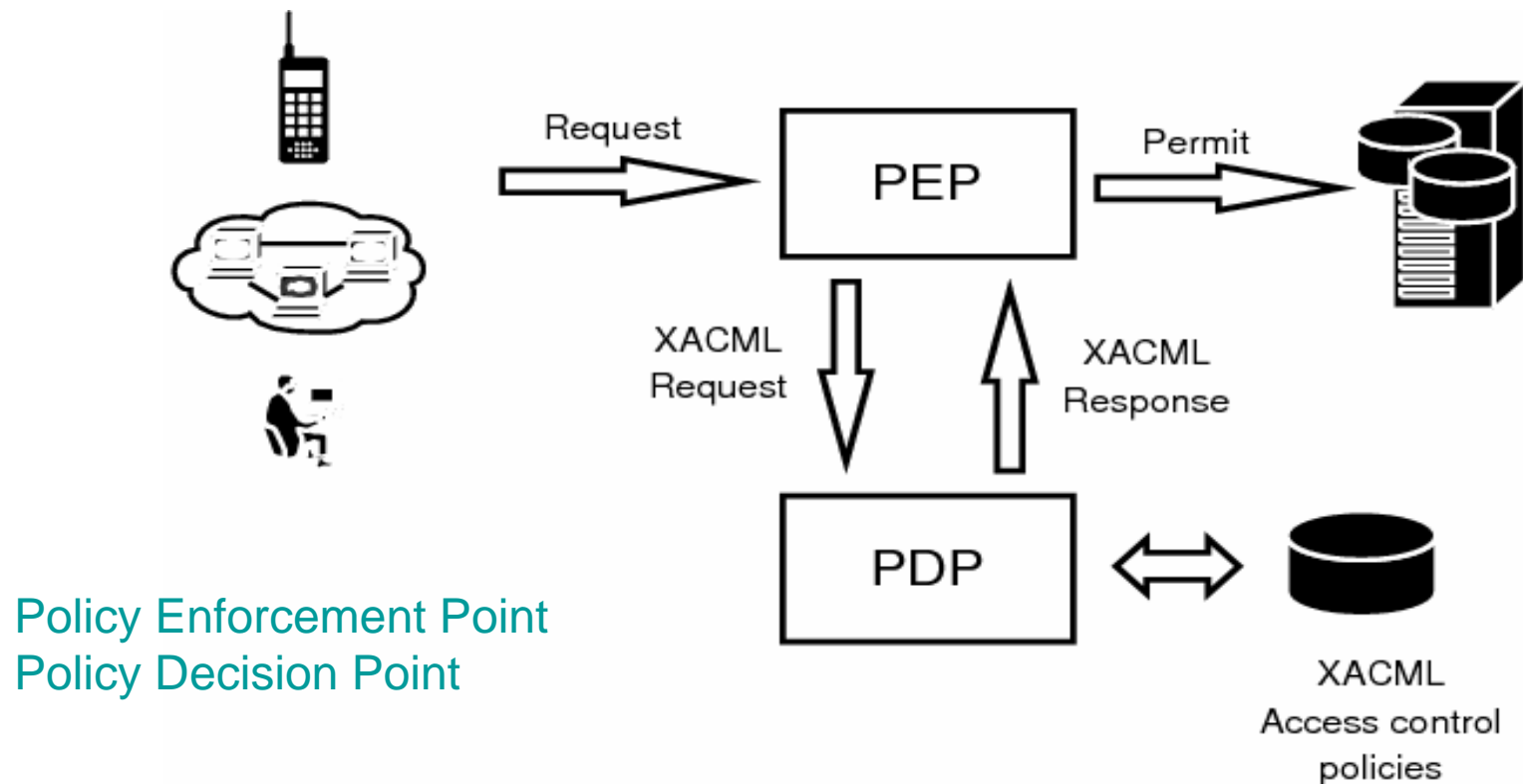
Action

Resource

- A policy
 1. A professor **can** read or modify the file of course marks
 2. A student **can** read the file of course marks
 3. A student **cannot** modify the file of course marks
- Question:
 - A subject that is both student and professor wants to modify the file of course marks
 - Will his request be accepted or refused?
- Users and administrators should know about these potential inconsistencies
 - avoid security leaks, denial of service and unauthorized access

XACML overview

- eXtensible Access Control Markup language : an OASIS standard
- Architecture, policies and messages

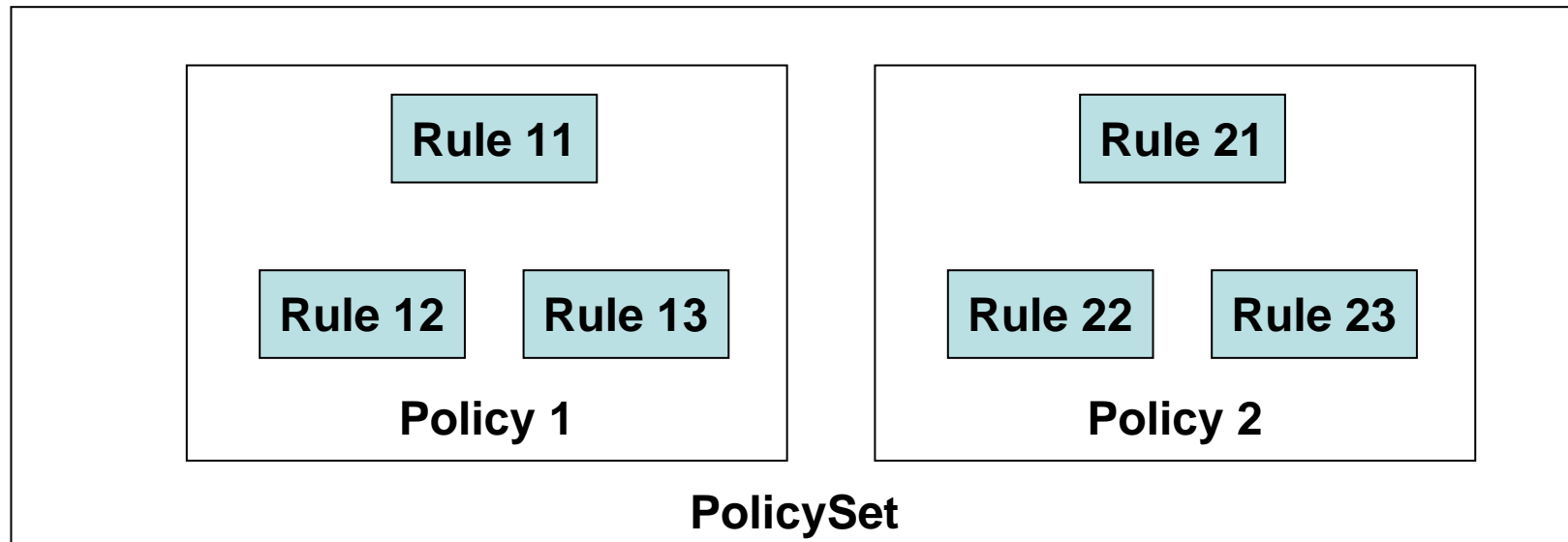


XACML Request

```
<Request>
  <Subject>
    <Attribute AttributeId="Role" DataType="string">
      <AttributeValue>Professor</AttributeValue>
    </Attribute>
  </Subject>
  <Resource>
    <Attribute AttributeId="ResourceName" DataType="string">
      <AttributeValue>CourseMarksFile</AttributeValue>
    </Attribute>
  </Resource>
  <Action>
    <Attribute AttributeId="ActionName" DataType="string">
      <AttributeValue>Read</AttributeValue>
    </Attribute>
  </Action>
  <Environment/>
</Request>
```

XACML Structures

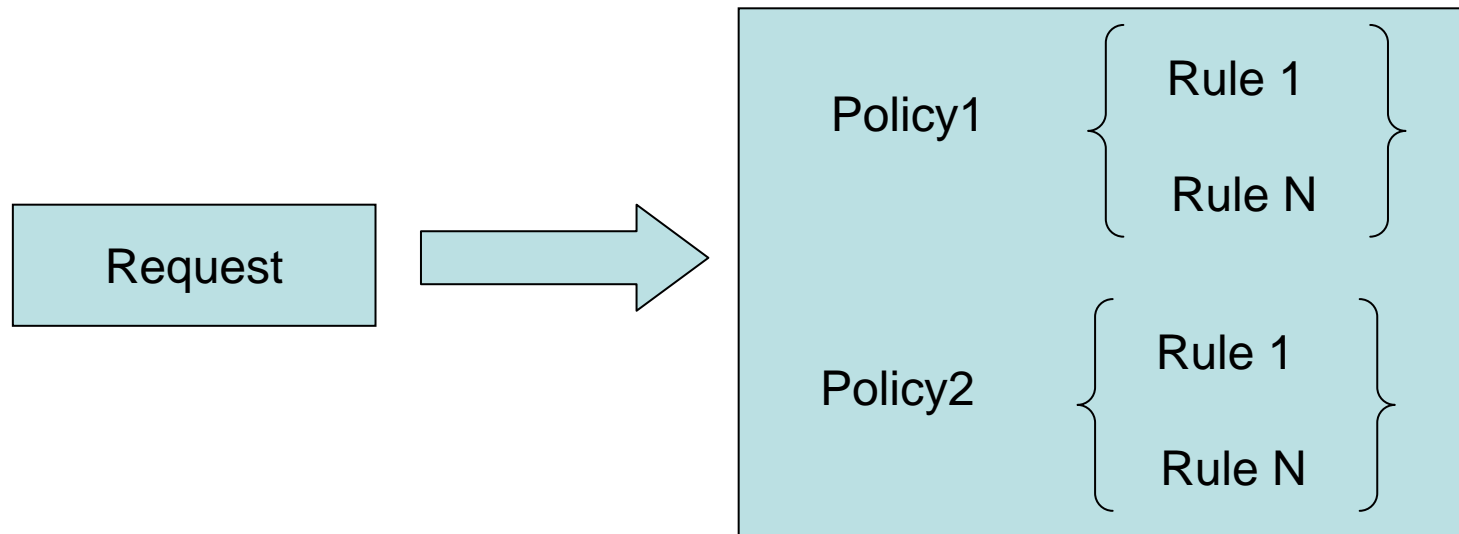
- A syntax based on XML to define Access Control
 - Rules
 - Policies
 - Policy sets



An XACML policy

```
<Policy PolicyId="OperationFichiersNotes" RuleCombiningAlgId="deny-overrides">  
  <Description>  
    Access to course marks file  
  </Description>  
  <Target>  
    <Subjects><AnySubject/></Subjects>  
    <Resources><AnyResource/></Resources>  
    <Actions><AnyAction/></Actions>  
  </Target>  
  <Rule RuleId="Rule1" Effect="Permit">  
    <Description>A professor can read or modify the course marks file</Description>  
    <Target>  
      <Subjects>  
        <Subject>  
          <SubjectMatch MatchId="string-equal">  
            <AttributeValue DataType="string">Professor</AttributeValue>  
            <SubjectAttributeDesignator AttributeId="Role" DataType="string"/>  
          </SubjectMatch>  
        </Subject>  
      </Subjects>  
    </Target>  
  </Rule>  
</Policy>
```

Targets and Conditions



- Not all policies are applied to a request
- Targets define the applicability of policy sets, policies and rules
- Conditions are additional and more complex filters for rules

Targets

- A policy
 1. A professor **can** read or modify the file of course marks
 2. A student **can** read the file of course marks
 3. A student **cannot** modify the file of course marks
- Rule 2 is applied when (target)
 - Subject's role is "student"
 - Resource's name is "course marks"
 - Action's name is "read"
- Request : a student Bob wants to read the file of course marks
 - Rule 2 is applied but not Rule1 nor Rule 3

Target

```
<Rule RuleId="Rule2" Effect="Permit">
```

```
  <Description>A student can read the course marks file</Description>
```

```
  <Target>
```

```
    <Subjects>
```

```
      <Subject>
```

```
        <SubjectMatch MatchId="string-equal">
```

```
          <AttributeValue DataType="string">Student</AttributeValue>
```

```
          <SubjectAttributeDesignator AttributeId="Role" DataType="string"/>
```

```
        </SubjectMatch>
```

```
      </Subject>
```

```
    </Subjects>
```

```
    <Resources>
```

```
      <Resource>
```

```
        <ResourceMatch MatchId="string-equal">
```

```
          <AttributeValue DataType="String">CourseMarksFile</AttributeValue>
```

```
          <ResourceAttributeDesignator AttributeId="ResourceName" DataType="String"/>
```

```
        </ResourceMatch>
```

```
      </Resource>
```

```
    </Resources>
```

```
    <Actions>
```

```
      <Action>
```

```
        <ActionMatch MatchId="string-equal">
```

```
          <AttributeValue DataType="string">Read</AttributeValue>
```

```
          <ActionAttributeDesignator AttributeId="ActionName" DataType="string"/>
```

```
        </ActionMatch>
```

```
      </Action>
```

```
    </Actions>
```

```
  </Target>
```

```
</Rule>
```

Subject

Resource

Action

Combining Algorithms

- Mechanisms to resolve conflicts online
- Example:
 - Bob is PhD student and an assistant professor,
 - he wants to modify the file of course marks
- Permit-overrides : Permit
- Deny-Overrides : Deny
- First-Applicable : Permit (Rule 1 appears before Rule 3 in an xml file)
- Only-one-applicable : Indeterminate (Error)

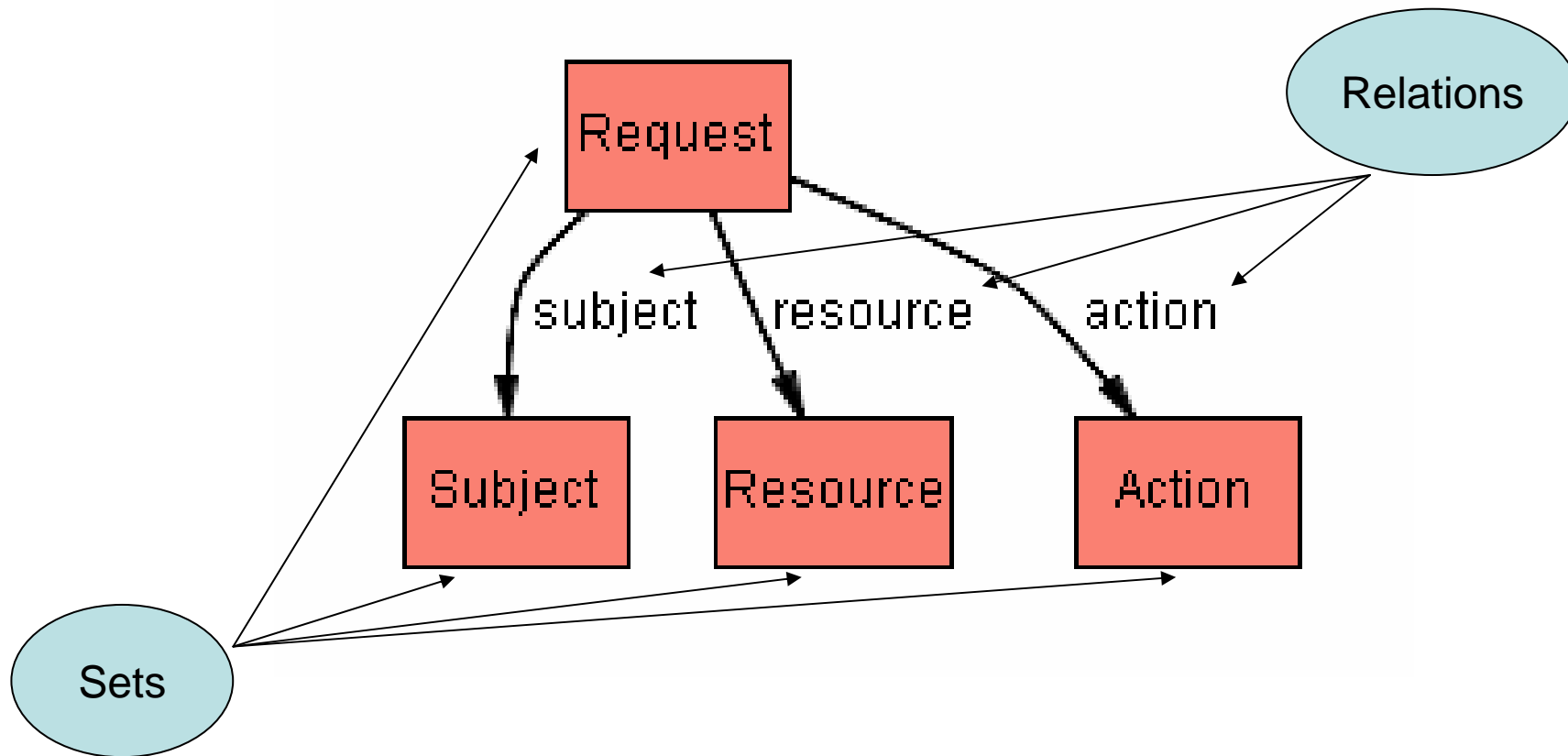
A Logical Model of XACML

- Use of sets, relations and functions
- Structures and constraints
- use of Alloy syntax
- Alloy
 - Modeling language
 - Analyzer tool
 - Relational first-order logic

Alloy

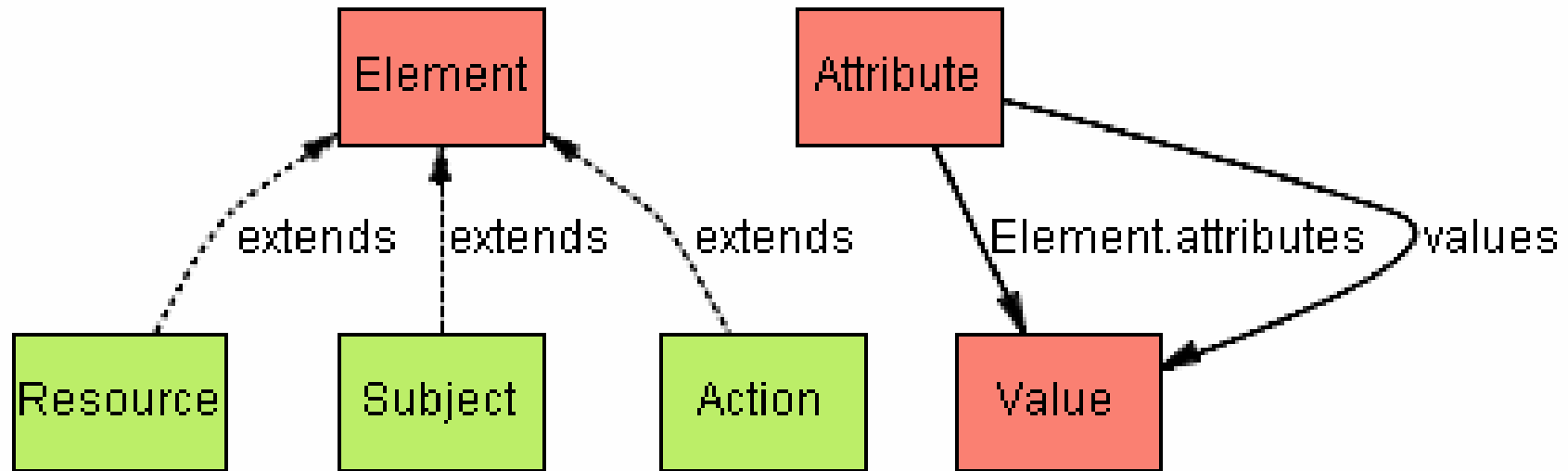
- Structural
 - Signature
 - Relation
- Declarative
 - first-order logic
 - facts, predicates, functions, and assertions
- Analyzable
 - Simulation and automatic verification
 - run predicate
 - check assertion

Examples: Request



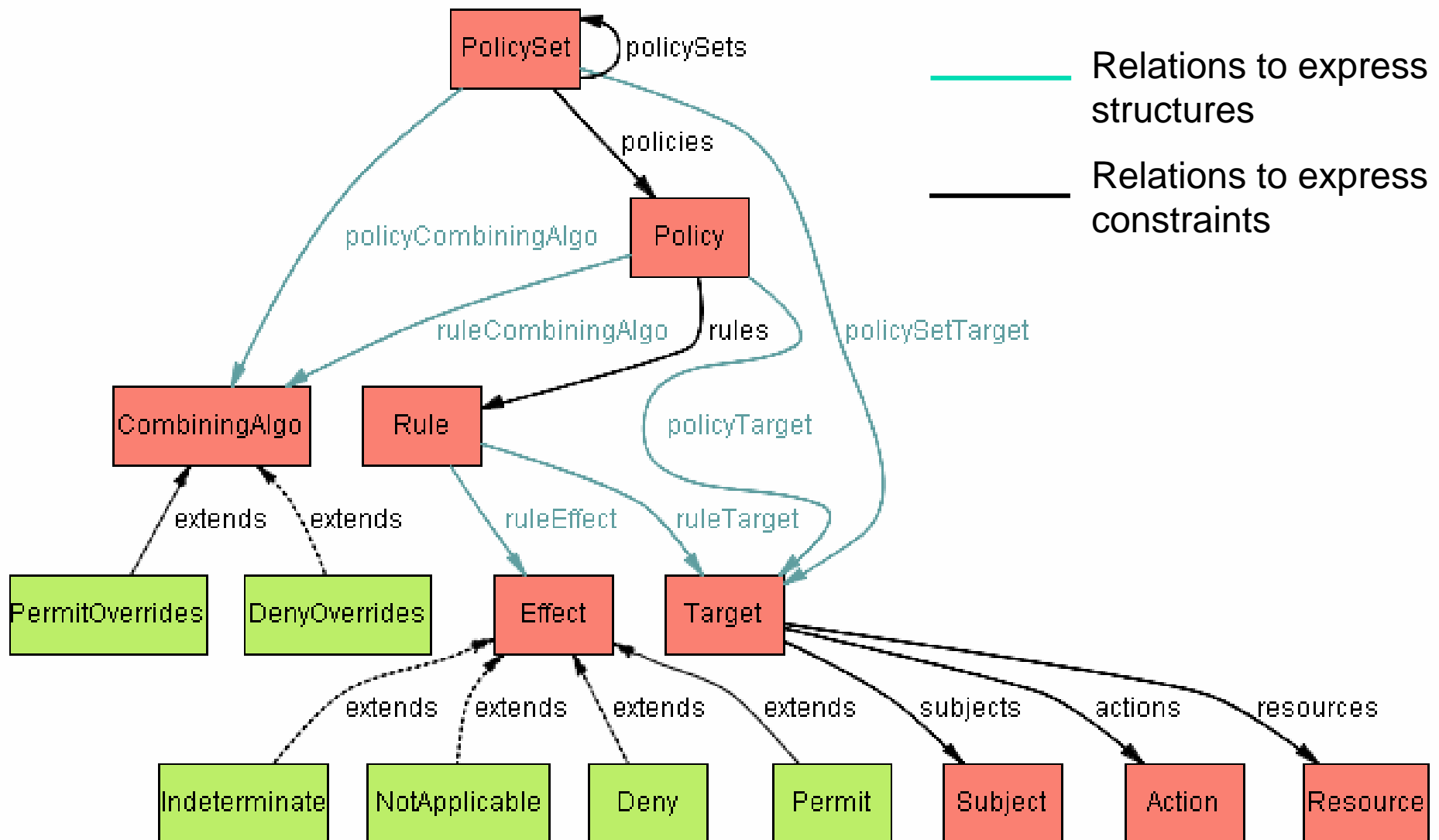
Basic structures

- Relations
 - values : Attribute \rightarrow Value : defines possible values for an attribute
 - attributes : Element \rightarrow Attribute \rightarrow Value : defines the actual values for an attribute
- Resources, subjects and actions are elements defined by a set of valued attributes



Inheritance as subsetting

Structures



Constraints

- Use of functions and predicates
- First order logic

Constraints

- a predicate that evaluates a request against a target to check whether the target matches the request

```
pred targetMatch (t : Target, r : Request) {  
  some e: t.subjects | elementMatch(r.subject, e)  
  some e: t.resources | elementMatch(r.resource, e)  
  some e: t.actions | elementMatch(r.action, e)  
}
```

Constraints

- A function that returns the response of a given rule regarding a given request

```
fun ruleResponse (r : Rule, req : Request) : Effect {  
  if targetMatch(r.ruleTarget, req) then r.ruleEffect  
  else NotApplicable  
}
```

Combining Algorithms

```
fun rulePermitOverrides ( p : Policy, req : Request) : Effect {  
  if existPermit(p,req) then Permit  
  else if existDeny(p,req) then Deny  
  else NotApplicable  
}  
fun ruleDenyOverrides ( p : Policy, req : Request) : Effect {  
  if existDeny(p,req) then Deny  
  else if existPermit(p,req) then Permit  
  else NotApplicable  
}
```

Verification and Validation

- Check properties
- Use of predicates and assertions
- Examples
 1. An example of a rule returning a permit response regarding a specific request → an example?
 2. Inconsistency: different rules within the same policy return different decisions (permit and deny) → an example?
 3. Access should always be granted to a professor requesting modification → a counterexample?

Access Control Policy

- Rule1 :
 - A professor can read or modify the file of course marks
- Rule2 :
 - A student can read the file of course marks
- Rule3 :
 - A student cannot modify the file of course marks

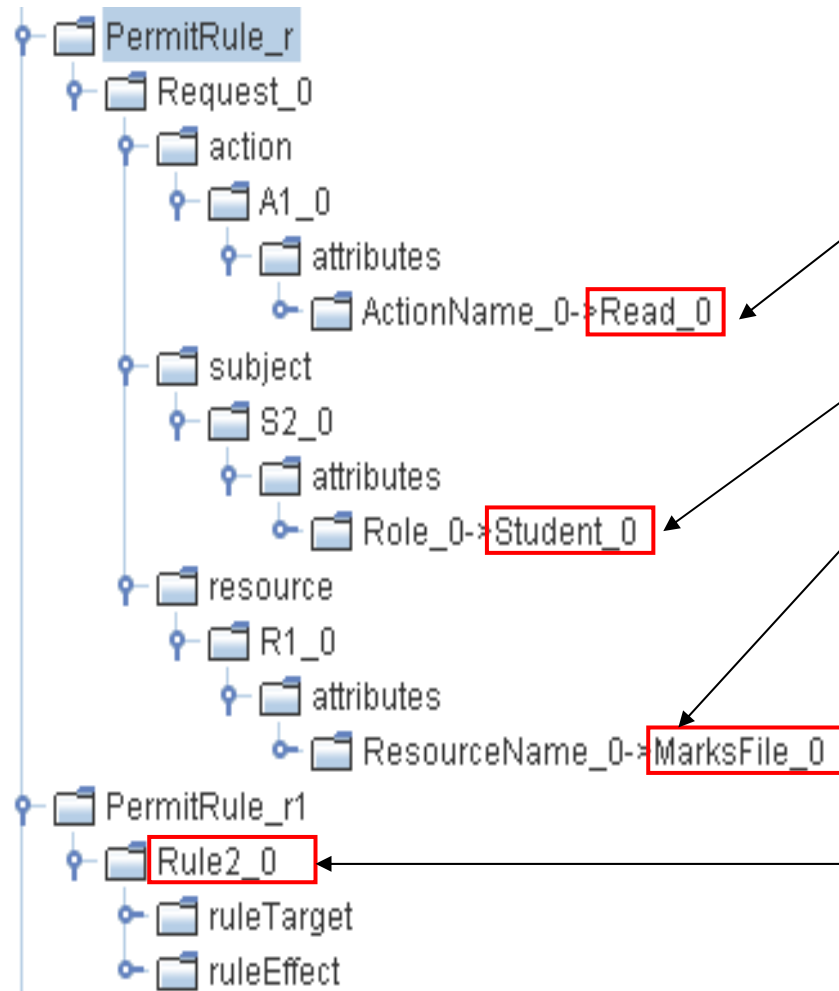
Example 1

- An example of a rule returning a permit response regarding a specific request

```
pred PermitRule(q : Request, r : Rule){  
  ruleResponse(r,q) = Permit  
}
```

```
run PermitRule for 8 but 1 Request
```

Example 1



- When

- A Read access request from
- A students
- On course marks file
- Rule2 is applied and returns a permit

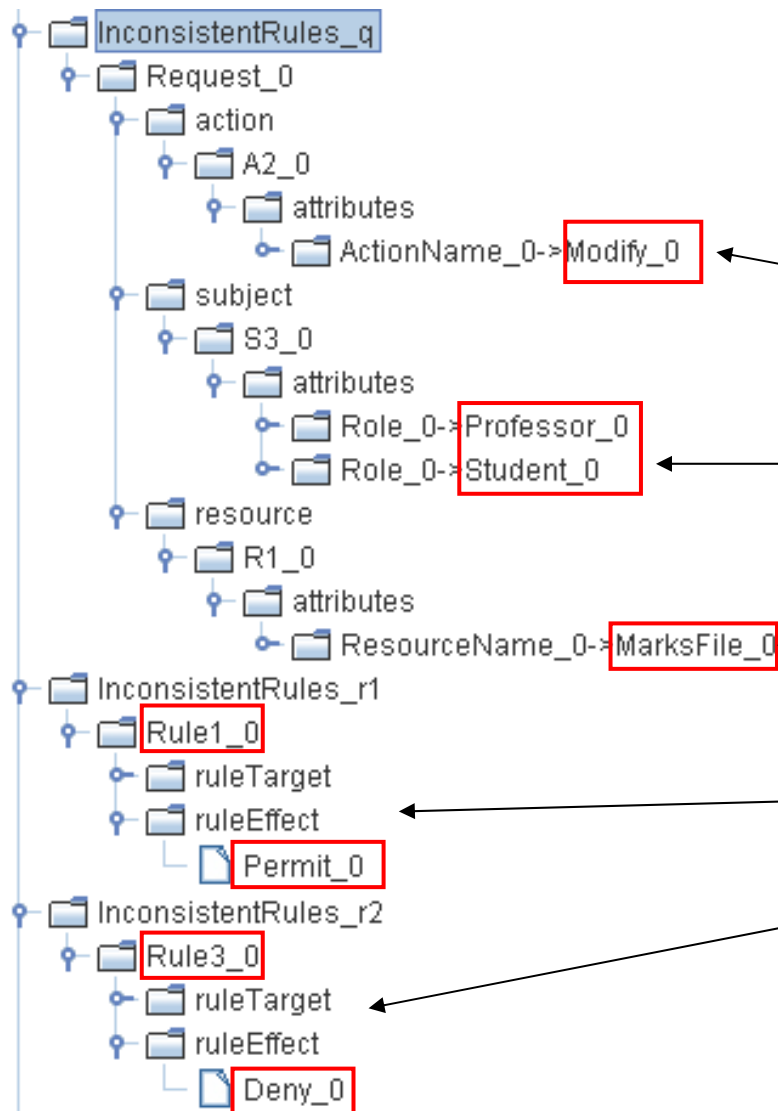
Example 2

- Inconsistency: different rules within the same policy return different decision (permit and deny)

```
pred InconsistentPolicy (p : Policy, req : Request) {  
    some r : p.rules | ruleResponse(r, req) = Permit  
    some r : p.rules | ruleResponse(r, req) = Deny  
}
```

```
run InconsistentPolicy for 8 but 1 Request
```

Example 2



• Both rule1 and rule3 are applied when

– A modification request comes from

– A subject with both professor and student role

– On the file of course marks

– Rule1's response is permit

– Rule3's response is deny

Example 3

- Access should always be granted to a professor (and not student) requesting modification

```
assert PermitForProfessor {  
  all q : Request {  
    {~(q.subject.attributes).Attribute = Professor}  
    => policyResponse(P,q) = Permit } }
```

check PermitForProfessor for 8 but 1 Request

- Alloy doesn't find any solution

Related work

- MTBDDs to verify XACML policies
- Conflicts detection tools for PONDER
- RW → verification → XACML
- Other logical approaches

Conclusion

- XACML validation and verification using model-checking and first-order logic
- Only a subset of XACML was covered
- A translation tool for transforming XACML policies to Alloy specifications

Future work

- GUI to permit clear visualization of XACML rules
 - More intuitive syntax than XACML
- GUI to permit editing XACML
 - Without touching XACML code directly
- GUI to display the results of the analysis in user-friendly format
 - Immediately after editing