Science of Computer Programming 232 (2024) 103046

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original Software Publication

XACML2mCRL2: Automatic transformation of XACML policies into mCRL2 specifications

Hamed Arshad [a,*], Ross Horne [b], Christian Johansen [c], Olaf Owe [a], Tim A.C. Willemse [d]

[a] University of Oslo, Oslo, Norway
[b] Computer and Information Sciences, University of Strathclyde, Glasgow, United Kingdom
[c] Norwegian University of Science and Technology, Gjøvik, Norway
[d] Eindhoven University of Technology, Eindhoven, Netherlands

ARTICLE INFO

Keywords:
XACML
mCRL2
Formal verification
Access control

ABSTRACT

The eXtensible Access Control Markup Language (XACML) is a popular OASIS standard for the specification of fine-grained access control policies. However, the standard does not provide a proper solution for the verification of XACML access control policies before their deployment. The first step for the formal verification of XACML policies is to formally specify such policies. Hence, this paper presents XACML2mCRL2, a tool for the automatic translation of XACML access control policies into mCRL2. The mCRL2 specifications generated by our tool can be used for formal verification of important properties of access control policies, such as completeness or inconsistency, using the well-known mCRL2 toolset.

Code metadata

Code metadata description	
Current code version	v1.3
Permanent link to repository	https://github.com/ScienceofComputerProgramming/SCICO-D-22-00293
Link to code	Follow the path: /XACML2mCRL2/Source Code/
Permanent link to Reproducible Capsule	https://www.doi.org/10.24433/CO.2697126.v1
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	Java 14, XSLT 1.0, Eclipse IDE (4.17.0)
Compilation requirements, operating environments and dependencies	Microsoft Windows, Linux, macOS
Link to developer documentation/manual	Follow the path: /XACML2mCRL2/User Manual/
Support email for questions	hamedarshad@aol.com

The code (and data) in this article has been certified as Reproducible by Code Ocean: https://codeocean.com/. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: hamedar@ifi.uio.no, hamedarshad@aol.com (H. Arshad), ross.horne@strath.ac.uk (R. Horne), christian.johansen@ntnu.no (C. Johansen), olaf@ifi.uio.no (O. Owe), T.A.C.Willemse@tue.nl (T.A.C. Willemse).

https://doi.org/10.1016/j.scico.2023.103046
Received 13 November 2022; Received in revised form 15 October 2023; Accepted 16 October 2023

Available online 20 October 2023

1. Motivation and significance

The eXtensible Access Control Markup Language (XACML) standard [1] provides a declarative policy language with a large collection of features that allow the specification of fine-grained, attribute-based access control policies. Such policies determine who can access what and under what conditions, e.g., when, from where, and how. Although XACML is popular both in academia and industry, it does not cover the problem of verifying the completeness and consistency of the specified access control policies. The XACML standard does contain, what is called, combining algorithms for resolving at runtime the conflicts/inconsistencies between policies/rules. For instance, the *DenyOverrides* combining algorithm selects the `Deny` effect when a request results in both `Permit` and `Deny`. The XACML combining algorithms are manageable for a small number of policies maintained locally. However, for complex real-world policies or when integrating policies from different domains/organizations the implications of these combining algorithms can be difficult to comprehend. Inconsistent and incomplete policies may permit unauthorized access, which breaches confidentiality, or may deny the access of a legitimate user, which endangers availability. For example, a conflict in policies (with *DenyOverrides* taking effect) could be life-threatening if it denies the hospital emergency staff access to the medical records of a patient.

Formal verification could be used to ensure properties such as the consistency of XACML access control policies before their actual deployment. To be able to handle the verification of XACML realistic policies in context we have chosen in [2] to work with a mature verification framework called mCRL2 [3–5] offering us the following.

i) mCRL2 allows us to define custom data types to be able to handle the various types of values allowed by XACML.
ii) mCRL2 comes with a powerful toolset [6], publicly available, including formal verification functionalities such as model checking and generating counterexamples.
iii) The process algebraic language of mCRL2 supports parallel composition used for specifying distributed systems. This allows us to verify the access control policies *in context*, i.e., within systems that are going to be protected by such policies, by modeling also the system and putting it in parallel with the model of the policy.

The first step for formal verification of XACML policies using mCRL2 is to formally specify them in mCRL2. However, specifying XACML policies in mCRL2 requires special expertise that security administrators who author XACML policies normally do not have. To tackle this problem, this paper presents the XACML2mCRL2 tool, which automatically transforms XACML policies into mCRL2 specifications. Policy authors can get the mCRL2 specifications of their XACML policies using our tool (the code metadata is listed in table on page 1), and then analyze and verify them using the mCRL2 toolset [6]. In other words, the addition of our tool to the mCRL2 toolset allows security administrators, who do not have knowledge about formal methods, to formally verify their XACML access control policies easily.

2. Illustrative examples

This section provides an example to demonstrate how XACML access control policies can be formally verified using our tool and the mCRL2 toolset. More examples can be found in [2].

Assume that in a healthcare institution, there is a requirement stating that patients' `MedicalRecords` should only be accessible to `Doctors`. One may address this requirement by defining **Rule A**, which states that if the requester (i.e., `subjectid`) is not a `Doctor` and the requested object (i.e., `resourceid`) is `MedicalRecords`, the request should be rejected, i.e., the response is `Deny`.

Rule A: $(resourceid = MedicalRecords) \wedge \textbf{NOT}(subjectid = Doctor) \Rightarrow Deny$

The emergency department may also require another rule giving `Read` access to `MedicalRecords` to `EmergencyStaff`, which can be addressed by adding **Rule B** to the `PolicySet`.

Rule B: $(resourceid = MedicalRecords) \wedge (subjectid = EmergencyStaff) \wedge (actiontid = Read) \Rightarrow Permit$

Fig. 1 represents the XACML version of these two rules, which are included in a `Policy` named *PolicySOP* within a *root* `PolicySet`. The annotation with shapes, colors, and numbered labels is used later in Fig. 2 to show the corresponding mCRL2 parts of the specifications generated. For instance, in Fig. 1, the *root* PolicySet, the *PolicySOP* Policy, *RuleA*, and *RuleB*, labeled 1 to 4, are translated into four different mCRL2 processes with identical names as labeled 1 to 4 in Fig. 2. The Targets of *RuleA* and *RuleB* (labeled 3.1 and 4.1 in Fig. 1) are translated to corresponding conditions and used in the mCRL2 process of their parent policy, which is *PolicySOP* Policy (labeled 3.1 and 4.1 in Fig. 2). Furthermore, the Condition parts of *RuleA* and *RuleB*, labeled 3.2 and 4.2 in Fig. 1, are translated and used in their respective mCRL2 processes, which have the same labels in Fig. 2. As represented in Fig. 2, the Effects of *RuleA* and *RuleB*, which are *Deny* and *Permit*, respectively (labeled 3.3 and 4.3 in Figs. 1 and 2), are encoded into the Response actions of the corresponding mCRL2 processes.

The generated mCRL2 specifications can now be analyzed using the mCRL2 toolset (we used the mCRL2 IDE version 202106.0, Release). To do so, we create a project (here we name it "SOP") and add the mCRL2 specifications into the main window. To verify the consistency, we add the Policy-Consistency property (as shown in Fig. 3), which is a generic property for XACML policies introduced in [2] and reproduced below.

Fig. 1. A simplified XACML policy. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Property *(Policy-Consistency).* A set of policies is conflict-free if there is no inconsistency between policies.

```
forall rs:FSet(SAtt), ro:FSet(OAtt), ra:FSet(AAtt).[Request(rs, ro, ra)]
!(<true*.Response(rs, ro, ra, Deny)> true && <true*.Response(rs, ro, ra, Permit)> true)
```

```
sort SAtt      = struct attribute(name:SAttName, value:SAttValue);
sort SAttName  = struct subjectid;
sort SAttValue = struct Doctor|EmergencyStaff;
sort OAtt      = struct attribute(name:OAttName, value:OAttValue);
sort OAttName  = struct resourceid;
sort OAttValue = struct MedicalRecords;
sort AAtt      = struct attribute(name:AAttName, value:AAttValue);
sort AAttName  = struct actionid;
sort AAttValue = struct Read;
sort Decision  = struct Permit | Deny;
act
    Request:FSet(SAtt)#FSet(OAtt)#FSet(AAtt);
    Response:FSet(SAtt)#FSet(OAtt)#FSet(AAtt)#Decision;
proc
1 PolicySet_root(RS:FSet(SAtt), RO:FSet(OAtt), RA:FSet(AAtt)) =
      Policy_PolicySOP(RS,RO,RA);
2 Policy_PolicySOP(RS:FSet(SAtt), RO:FSet(OAtt), RA:FSet(AAtt))=
3.1 ((attribute(resourceid, MedicalRecords) in RO)
    )-> Rule_RuleA(RS,RO,RA)
    +
4.1 (  (attribute(resourceid, MedicalRecords) in RO)
    &&
       (attribute(actionid, Read) in RA)
    )-> Rule_RuleB(RS,RO,RA);
3 Rule_RuleA(RS:FSet(SAtt), RO:FSet(OAtt), RA:FSet(AAtt))=
3.2 !(attribute(subjectid, Doctor) in RS) -> Response(RS,RO,RA,Deny);
4 Rule_RuleB(RS:FSet(SAtt), RO:FSet(OAtt), RA:FSet(AAtt))=
4.2 (attribute(subjectid, EmergencyStaff) in RS) -> Response(RS,RO,RA,Permit);
init sum RS:FSet(SAtt).sum RO:FSet(OAtt).sum RA:FSet(AAtt).(RS !={} && RO !={} &&
RA !={})-> Request(RS,RO,RA).PolicySet_root(RS,RO,RA);
```

Fig. 2. The generated mCRL2 specifications for the XACML policy in Fig. 1.

After pressing the button for verification of all properties (as in Fig. 4), any properties not respected by the specifications will be highlighted with red as shown in Fig. 4. Clicking on the "C" button in the "Properties" window will show a counterexample violating the property. In this example, there is a conflict between **Rule A** and **Rule B** as there are two Response actions with different decisions, i.e., *Permit* and *Deny*, after the Request action. In other words, both **Rule A** and **Rule B** that have different *Effects* cover the same request, Request({*attribute*(*subjectid*, *EmergencyStafff*)}, {*attribute*(*resourceid*, *MedicalRecords*)}, {*attribute*(*actionid*, *Read*)}). Counterexamples help policy authors to detect where the problems are in their policies and how to resolve them before the actual deployment of the policies.

3. Software description

Our tool is an open-source tool that uses the eXtensible Stylesheet Language Transformations (XSLT) (version 1.0) [7] for transforming XACML policies written in XML, into mCRL2 specifications, based on XSLT transformation rules defined in a separate file named `policy.xsl`. This file is the first main component of XACML2mCRL2, the second being a user interface developed in Java. Because of this separation, one can also use (instead of our Java user interface) for processing the XSLT policy transformation rules other XSLT processors, such as *xsltproc*,[1] which is a command line XSLT processor based on the free *libxslt* library.[2]

[1] macOS and various Linux distributions feature *xsltproc* by default. It is also possible to install it on Windows.

[2] Released under the MIT License.

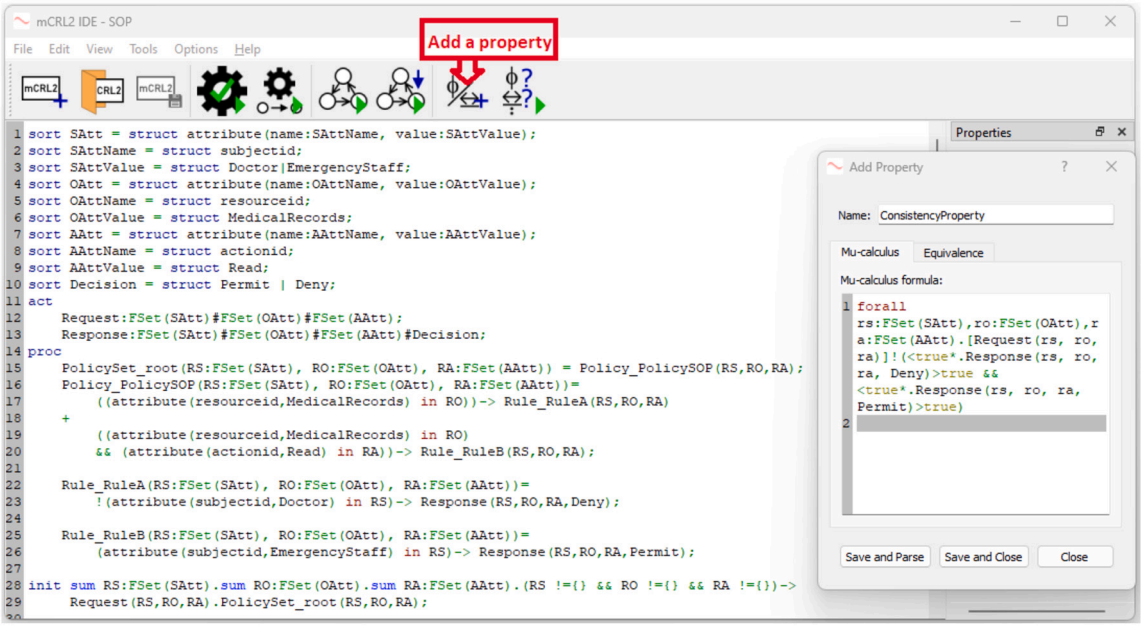

Fig. 3. Adding the specifications and a property to a project in mCRL2 IDE.

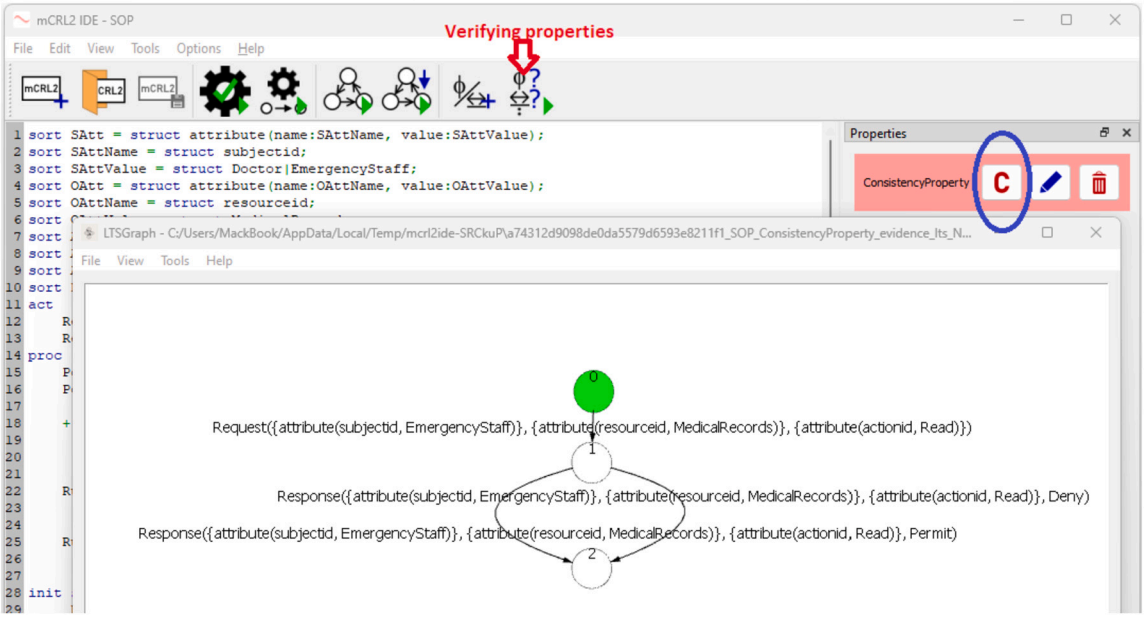

Fig. 4. A counterexample generated by the mCRL2 IDE.

Our tool (particularly, the transformation rules we defined in `policy.xsl`) covers and translates all components and elements of the XACML policy model, except for the XACML combining algorithms. These algorithms are designed to resolve conflicts between policies at runtime. While this solution may prove useful for a small number of policies and rules defined by a single individual, it becomes challenging to select an appropriate combining algorithm when several policies are defined by different individuals in real-world applications. The situation worsens when integrating policies from different domains, as policy authors in those domains are unlikely to be aware of policies from other domains at the time of authoring policies. Even assuming one successfully selects an appropriate combining algorithm, unexpected behaviors can still arise due to interactions between different policies or rules. For these reasons, we focus on verifying the consistency and completeness of policies at the time of authoring (i.e., static analysis). By ensuring that policies are correct and free of conflicts before they are deployed, many of the issues that might arise during runtime can be prevented. This approach provides greater confidence in the behavior of policy-based systems while also reducing the burden

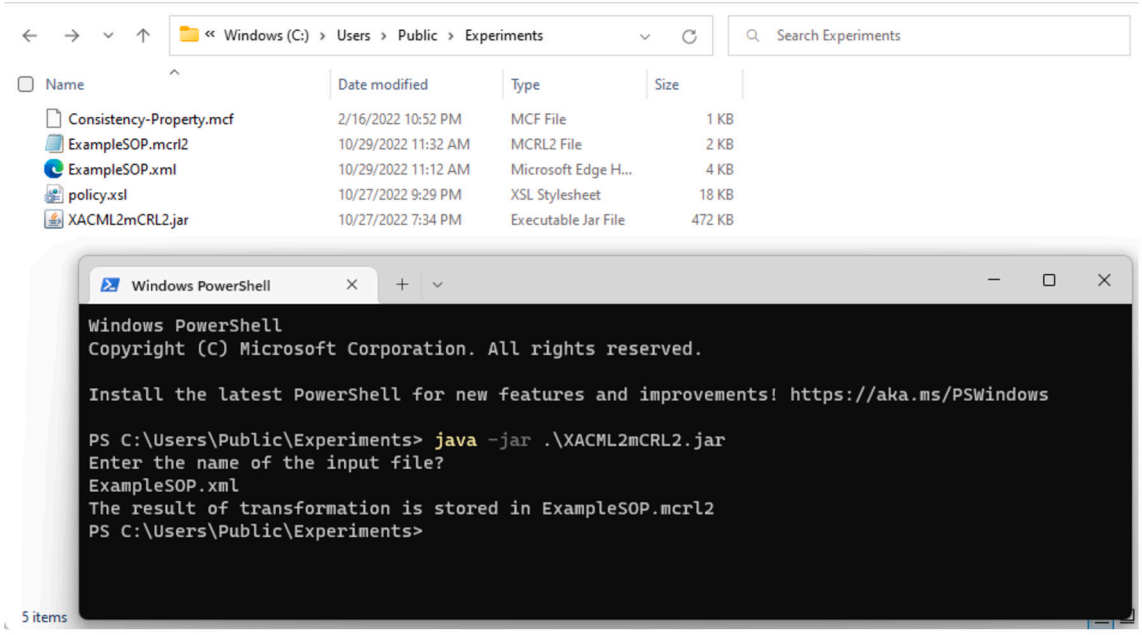

Fig. 5. Running XACML2mCRL2.

on policy authors and security administrators. Therefore, considering combining algorithms would go against the main purpose of formally verifying XACML policies, which is to find and address inconsistencies before the actual deployment of the policies. In the same vein, our tool does not translate `Advices` because they are optional (i.e., can be ignored by the enforcement point) and thus are not relevant for formal verification.

The XACML standard (XACML v3.0) defines four evaluation results: `Permit`, `Deny`, `NotApplicable`, and `Indeterminate`. However, in real-world scenarios, we are often interested in only `Permit` and `Deny` results. Both `NotApplicable` (when policies do not cover the given request) and `Indeterminate` (when a decision cannot be made due to missing or conflicting information, errors in policy evaluation, or external factors) provide a way to communicate (at runtime) to the requester that the authorization decision could not be made for specific reasons. However, our tool aims to provide a simple and easy-to-use solution for policy verification statically and at the time of creation rather than runtime decision-making. The policy decision point (PDP) in XACML-based systems can still return `NotApplicable` and `Indeterminate` results. However, static analysis of policies, i.e., formal verification of policies before their actual deployment, helps to identify incomplete policy sets (by verifying the Policy-Completeness property introduced in [2]), thus removing the need for the `NotApplicable` authorization result at runtime. Therefore, as a deliberate design decision aligning with our tool's focus and goals, we choose to disregard the `NotApplicable` and `Indeterminate` authorization results in our tool.

Our tool supports XACML policies that are specified based on the latest version of the XACML standard, i.e., XACML Version 3.0. XACML policies should not contain implicit information, and everything should explicitly exist in the policies. In other words, the values of attributes need to be explicitly specified using the `AttributeValue` element (and not using the `AttributeSelector` element, which can be used to provide an attribute value implicitly), and `PolicySets` and `Policies` should not be referenced in a `PolicySet` using, respectively, `PolicySetIdReference` and `PolicyIdReference` elements. Furthermore, every element of XACML policies must include all the `Required` elements and attributes according to the XACML standard. For instance, the `ObligationExpression` element contains `ObligationId`, `FulfillOn`, and `AttributeAssignmentExpression`, where the first two are `Required` and the last is `Optional` according to the XACML standard. Similarly, `Category`, `AttributeId`, and `MustBePresent` are `Required` in an `AttributeDesignator` element. Listing 4 in Appendix A demonstrates the implemented conditions for the enforcement of the transformation requirements.

The corresponding mCRL2 specifications of a set of XACML policies (rules) that are stored into an XML file can be obtained by running the XACML2mCRL2 tool using the command `java -jar XACML2mCRL2.jar` and providing the name of the XML file (Fig. 5). The tool generates the mCRL2 specifications and stores them into a file with `.mcrl2` extension, where its name is the same as that of the provided XML file.[3]

[3] To employ *xsltproc*, the following command can be used: `xsltproc policy.xsl InputNAME.xml > OutputNAME.mcrl2`.

Table 1
Analyzing policies generated using XACBench.

File Name	XACML			mCRL2		Transformation Time
	# of policies	# of rules	# of attributes	# of states	# of transitions	(Milliseconds)
Xbench5.xml	5	5	12	2838	6628	13.93
			13	14178	31168	
			14	59538	129328	
			15	240978	521968	
Xbench10.xml	10	10	12	2838	10340	24.41
			13	14178	48224	
			14	59538	199760	
			15	240978	805904	
Xbench20.xml	20	20	12	2838	17600	67.54
			13	14178	80600	
			14	59538	332600	
			15	240978	1340600	
Xbench40.xml	40	63	12	2838	51661	226.68
			13	14178	237766	
			14	59538	982186	
			15	240978	3959866	

4. Quality analysis

We evaluate the quality of our XACML2mCRL2 by testing it, in Subsection 4.1, using different sets of XACML policies,[4] and analyzing, in Subsection 4.2, sample code snippets.

4.1. Empirical results

We have previously used XACML2mCRL2 on the running examples presented in [2] and on others found on our GitHub repository [8]. These were meant to exemplify the correctness of XACML2mCRL2, i.e., how different XACML policies are transformed into proper mCRL2 specifications and then desired properties such as inconsistency, can be analyzed using the mCRL2 toolset; similarly to the example demonstrated in Section 2. Here we broaden the scope by translating real-world policies taken from the XACML 2.0 conformance tests converted to XACML 3.0 syntax[5] and several policies generated using XACBench [9].

Our tool is used for transforming four different XACML policysets (generated using XACBench) into mCRL2 specifications. Though the examined policysets include different numbers of policies, rules, and attributes as listed in Table 1, our tool transformed all of them into mCRL2 specifications very quickly. We conducted the transformation process for all four different XACML policysets to mCRL2 a total of 100 times to obtain the average transformation time. The experiments were conducted on a machine with an Intel Core i7-8550U CPU running at 1.80 GHz, 32 GB RAM, and Ubuntu 16.04 LTS (64-bit). According to the results presented in Table 1, the average transformation time for the first policyset, which consisted of 5 policies and 5 rules, was 13.93 milliseconds. For the second policyset with 10 policies and 10 rules, the average transformation time was 24.41 milliseconds. In the case of the third policyset, containing 20 policies and 20 rules, the average transformation time was 67.54 milliseconds. Lastly, for the fourth policyset consisting of 40 policies and 63 rules, the average transformation time was 226.68 milliseconds. The experiments revealed that the number of attributes does not impact the transformation time.

The results of analyzing the generated mCRL2 specifications using the mCRL2 toolset, as demonstrated in Table 1, show that the number of policies and rules inside a `PolicySet` has a direct effect on the number of transitions in the mCRL2 model. However, an increase in the number of attributes increases the number of both states and transitions in the mCRL2 model, which in turn the time required for the verification of desired properties (using the mCRL2 toolset) would be affected.

To evaluate the coverage of XACML2mCRL2, 452 different policies (some including only one rule) from the XACML 2.0 conformance tests (which are upgraded to XACML 3.0 syntax) are transformed into mCRL2 specifications. The current version of XACML2mCRL2 (i.e., version 1.3) transforms about 63% of the conformance test policies into well-formed and valid mCRL2 specifications. One reason for not transforming all conformance test policies into proper mCRL2 specifications is that some of such policies do not include any attribute. In other words, some of them are without any targets and conditions like the one represented in Listing 1.

The rest of the conformance test policies that were not transformed into proper mCRL2 specifications include complex condition parts with XACML functions that are not supported in the current version of XACML2mCRL2. Therefore, future versions of XACML2mCRL2 should focus on implementing more of these specific functions used in the conditions, such as working with numbers or sets as mCRL2 has support for arithmetic and set operations.

[4] All examples and examined policies exist on the XACML2mCRL2's GitHub repository.
[5] https://bit.ly/3EtglrC.

```
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     PolicyId="urn:oasis:names:tc:xacml:2.0:conformance-test:IIB001:policy" RuleCombiningAlgId="urn:oasis:names:tc:xacml
     :3.0:rule-combining-algorithm:deny-overrides" Version="1.0">
<Target/>
<Rule Effect="Permit" RuleId="urn:oasis:names:tc:xacml:2.0:conformance-test:IIB001:rule">
</Rule>
</Policy>
```

Listing 1: A policy (IIB001 in the XACML conformance tests) without any attributes.

```
<xsl:template name="TargetFunction">
 <xsl:if test="(xacml:Target!='')">
   (
  <xsl:for-each select = "xacml:Target/xacml:AnyOf/xacml:AllOf">
   <xsl:choose>
    <xsl:when test="position() != last()">
     (
      <xsl:call-template name="MatchFunction"/>
     )
     ||
     </xsl:when>
     <xsl:otherwise>
     (
      <xsl:call-template name="MatchFunction"/>
     )
     </xsl:otherwise>
   </xsl:choose>
  </xsl:for-each>
  )->
 </xsl:if>
</xsl:template>
```

Listing 2: XSLT code for transforming the Target of a Rule, Policy, or PolicySet.

```
<xsl:template name="MatchFunction">
 <xsl:param name="attname"/>
 <xsl:for-each select = "xacml:Match">
  <xsl:choose>
   <xsl:when test="position() != last()">
    (attribute(<xsl:call-template name="getName"><xsl:with-param name="attname" select="translate(xacml:
         AttributeDesignator/@AttributeId,'-','')"/></xsl:call-template><xsl:value-of select = "$attname"/>, <xsl:call-
         template name="ValueOfAttribute"/>) in
    <xsl:call-template name="CategoryFunction"/> &amp;&amp;
   </xsl:when>
   <xsl:otherwise>
    (attribute(<xsl:call-template name="getName"><xsl:with-param name="attname" select="translate(xacml:
         AttributeDesignator/@AttributeId,'-','')"/></xsl:call-template><xsl:value-of select = "$attname"/>,<xsl:call-
         template name="ValueOfAttribute"/>) in
    <xsl:call-template name="CategoryFunction"/>
   </xsl:otherwise>
  </xsl:choose>
 </xsl:for-each>
</xsl:template>
```

Listing 3: XSLT code for transforming Match elements in a Target element.

4.2. Sample code snippets analysis

XSLT transformation rules play the main role in transforming XACML policies into mCRL2. In order to make it easier to understand and update the XSLT transformation rules, different functions (i.e., XSLT templates) are defined for transforming different parts of an XACML policy, e.g., `target`, `condition`, `attributes`, and so on. For example, Listing 2 represents the XSLT code for translating the Target of a Rule, Policy, or PolicySet. This is a function (XSLT template) that can be called (re-used) several times by `<xsl:call-template name="TargetFunction"/>`. For instance, another function, i.e., `MatchFunction`, is called in lines 8 and 14 for translating `Match` elements that exist in the `Target` element.

Listing 3 shows that the `MatchFunction` calls another function, i.e., `<xsl:call-template name="CategoryFunction"/>` in lines 7 and 11, for checking the category of the attributes used in a `Match` element.

This kind of modularity (i.e., defining separate functions) makes it possible to easily update XACML2mCRL2 when there is a new version for the XACML standard. All the XSLT transformation rules required for transforming XACML policies into mCRL2

specifications are stored in the “policy.xsl” file, which needs to be in the same folder of the executable `jar` file (the developed user interface) when using XACML2mCRL2.[6]

5. Conclusions

In this paper, we have presented XACML2mCRL2, a tool that automatically generates corresponding formal specifications of XACML policies. The presented tool, which supports only the latest version of the XACML standard (i.e., XACML Version 3.0), translates all elements of XACML policies including obligations into mCRL2. The generated mCRL2 specifications can be analyzed/verified using the mCRL2 toolset, which is freely and publicly available. We have demonstrated that the combination of our XACML2mCRL2 tool and the mCRL2 toolset enables users to formally verify the XACML access control policies without knowing anything about formal methods. End users need to neither understand mCRL2 (and the generated mCRL2 specifications) nor define important properties of access control policies (as they are defined in [2]). The counterexamples that the mCRL2 toolset generates for violated properties help policy authors to detect and solve problems easily at the time of policy development. Formal verification of policies helps to achieve secure and reliable access control systems, which are fundamental security mechanisms.

The mCRL2 specifications generated by our tool can also be combined with the mCRL2 model of the systems that are supposed to be protected by such policies using the parallel composition offered by mCRL2 to perform *in-context* analysis. In-context verification of policies helps to detect and resolve any problem that they may cause in real-world systems.

6. Future plans

The current version of the software does not cover all the XACML offered functions that can be used in the condition part of rules to form more complex conditions, e.g., *less-than* or *greater-than*.[7] Making such updates does not require updating the Java part. Only the file containing the XSLT transformation rules needs to be updated.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Enforcement of transformation requirements

XACML2mCRL2 works based on a set of specific transformation requirements. If the input XACML policy fails to meet these requirements, it cannot be transformed into mCLR2, and instead, an error message will be displayed to the user. To ensure the proper enforcement of the transformation requirements explained in this paper, a series of conditions are implemented as shown in Listing 4.

```
1
2  <xsl:when test="((//xacml:Rule[not(.//xacml:AttributeDesignator)]) or (not($SubjectAtt = 'True') and not($ObjectAtt = '
       True') and not($ActionAtt = 'True') and not($EnvironmentAtt = 'True')))">
3   <xsl:text>
4    Error: The provided Policy includes a Rule that does not have any Attributes. In the current version of the tool,
         Rules should at least have an attribute.
5   </xsl:text>
6 </xsl:when>
7
8  <xsl:when test="(//xacml:AttributeSelector)">
9   <xsl:text>
10   Error: The provided Policy includes an AttributeSelector element.
11   The values of attributes need to be explicitly specified using the AttributeValue element (and not using the
         AttributeSelector element, which can be used to provide an attribute value implicitly).
12  </xsl:text>
13 </xsl:when>
14
15 <xsl:when test="(//xacml:PolicySetIdReference)">
16  <xsl:text>
17   Error: The provided Policy includes a PolicySetIdReference element.
18   The input policies should not contain implicit information, and everything should explicitly exist in the policies.
19   Therefore, PolicySets should not be referenced using a PolicySetIdReference element.
20  </xsl:text>
21 </xsl:when>
```

Listing 4. Conditions for enforcing transformation requirements.

[6] This is not a requirement when using *xsltproc* instead of the developed user interface.

[7] To further assist users, we have introduced a “Feature Inclusion - Work in Progress" section in the README file located in the tool’s GitHub repository. This section provides a list of XACML functions that are currently unsupported. The list will be regularly updated alongside new releases to ensure transparency and help users determine the availability of specific features.

```
<xsl:when test="(//xacml:PolicyIdReference)">
  <xsl:text>
    Error: The provided Policy includes a PolicyIdReference element.
    The input policies should not contain implicit information, and everything should explicitly exist in the policies.
    Therefore, Policies should not be referenced using a PolicyIdReference element.
  </xsl:text>
</xsl:when>

<xsl:when test="(//xacml:ObligationExpression[not(@ObligationId) or not(@FulfillOn)])">
  <xsl:text>
    Error: The provided Policy includes an ObligationExpression element that does not contain all the REQUIRED attributes
         according to the XACML standard.
    Please make sure that ObligationExpression elements contain both ObligationId and FulfillOn attributes that are
         REQUIRED attributes.
  </xsl:text>
</xsl:when>

<xsl:when test="(//xacml:AttributeDesignator[not(@AttributeId) or not(@Category) or not(@MustBePresent)])">
  <xsl:text>
    Error: The provided Policy includes an AttributeDesignator element that does not contain all the REQUIRED attributes
         according to the XACML standard.
    Please make sure that AttributeDesignator elements contain AttributeId, Category, and MustBePresent attributes that
         are REQUIRED attributes.
  </xsl:text>
</xsl:when>
```

Listing 4. (Continued.)

References

[1] B. Parducci, H. Lockhart, E. Rissanen, Extensible Access Control Markup Language (XACML) Version 3.0, OASIS Standard, 2013, pp. 1–154, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[2] H. Arshad, R. Horne, C. Johansen, O. Owe, T.A.C. Willemse, Process algebra can save lives: static analysis of XACML access control policies using mCRL2, in: FORTE, in: Lecture Notes in Computer Science, vol. 13273, Springer, 2022, pp. 11–30.

[3] J.F. Groote, M.R. Mousavi, Modeling and Analysis of Communicating Systems, MIT Press, 2014.

[4] J.F. Groote, J.J.A. Keiren, B. Luttik, E.P. de Vink, T.A.C. Willemse, Modelling and analysing software in mCRL2, in: FACS, in: Lecture Notes in Computer Science, vol. 12018, Springer, 2019, pp. 25–48.

[5] J.F. Groote, J.J.A. Keiren, Tutorial: designing distributed software in mCRL2, in: FORTE, in: Lecture Notes in Computer Science, vol. 12719, Springer, 2021, pp. 226–243.

[6] O. Bunte, J.F. Groote, J.J.A. Keiren, M. Laveaux, T. Neele, E.P. de Vink, W. Wesselink, A. Wijs, T.A.C. Willemse, The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: TACAS (2), in: Lecture Notes in Computer Science, vol. 11428, Springer, 2019, pp. 21–39.

[7] J. Clark, XSL Transformations (XSLT) Version 1.0, W3C Recommendation 1999, pp. 1–50, https://www.w3.org/TR/1999/REC-xslt-19991116.

[8] H. Arshad, R. Horne, C. Johansen, O. Owe, T.A.C. Willemse, GitHub repository for “Process algebra can save lives: static analysis of XACML access control policies using mCRL2”, https://github.com/haamedarshad/XACML2mCRL2, 2022.

[9] S. Ahmadi, M. Nassiri, M. Rezvani, XACBench: a XACML policy benchmark, Soft Comput. 24 (21) (2020) 16081–16096, https://doi.org/10.1007/s00500-020-04925-5.