A Scratch-based Graphical Policy Editor for XACML

Henrik Nergaard, Nils Ulltveit-Moe and Terje Gjøsæter
Institute of Information and Communication Technology, University of Agder, Jon Lilletuns vei 9, 4879 Grimstad, Norway

Keywords: Authorisation, XACML, Privacy, Editor, Smalltalk.

Abstract: This paper proposes a policy-maker-friendly editor for the eXtensible Access Control Markup Language (XACML) based on the programming language Scratch. Scratch is a blocks-based programming language designed for teaching children programming, which allows users to build programs like a puzzle. We take this concept one step further with an XACML policy editor based on the graphic programming elements of Scratch implemented in Smalltalk. This allows for aiding the user on how to build policies by grouping blocks and operators that fit together and also indicating which blocks that will stick together. It simplifies building the XACML policies while still having an XACML "feel" of the graphic policies.

1 INTRODUCTION

The eXtensible Access Control Markup Language (XACML) is a declarative access control policy language, standardised by OASIS, that is implemented in XML (Moses, 2005). The Standard defines a large set of XML elements, is very verbose and has high expressive power, which creates a high usage threshold for users that are not familiar with it or other XML based languages. Writing these polices can be difficult, especially for larger policies when the complexity increases, and user errors and typos can easily happen. XML lacks a user friendly representation, especially when the number of elements increases. Manual reading and error correction of bigger XML files can be a tedious and complicated task. The complexity in writing and correcting XACML policies may be some of the reason why simpler and less expressive authorisation standards (e.g. OAuth, RBAC or simple access control lists) may be preferred in practical implementations, or even that users decide to roll their own authorisation solution, with the possible security risks this may cause.

When creating a policy based on XACML, the creator has to have knowledge from both the standardisation of XACML as well as general XML behaviour. Problematic areas in XACML include the length of XACML attributes, correct handling of long URLs, and a vast amount of functions that must be used correctly, which is not always trivial for users. Without help from proper tools, creating large policies can involve tremendous work. Our solution is to use a user-friendly editor that helps in creating these XACML polices.

The rest of the article is organised as follows: Section 2 covers general background and motivation. Section 3 covers the design criteria and goals. In section 4, we describe the implementation of the graphical editor aimed at designing XACML policies. In section 5, the benefits and limitations of the proposed editor is discussed. Section 6 contains a summary of the article. Finally section 7 contains plans for future work on the editor.

2 BACKGROUND AND MOTIVATION

The policy editor for XACML policies is implemented based on the Scratch programming environment (Malan and Leitner, 2007; Resnick et al., 2009), on the Pharo Smalltalk engine(Bera and Denker, 2013). It aims at providing a policy-maker-friendly policy description language for designing XACML authorisation policies.It is also a design objective that it in the future shall be able to design XACML-based anonymisation policies for XML documents (Ulltveit-Moe and Oleshchuk, 2015; Ulltveit-Moe and Oleshchuk, 2012). Scratch is a programming language for children created by the Lifelong Kindergarten research group at Massachusetts Institute of Technology's Media LAB (Resnick et al., 2009),

Nergaard H., Ulltveit-Moe N. and Gjøsæter T..
A Scratch-based Graphical Policy Editor for XACML.
DOI: 10.5220/0005240101820190
In *Proceedings of the 1st International Conference on Information Systems Security and Privacy* (ICISSP-2015), pages 182-190
ISBN: 978-989-758-081-9

```
<PolicySet xmlns="&xacml;policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:gml="http://www.opengis.net/gml"
  xsi:schemaLocation="&xacml;policy cs-xacml-schema-policy-01.xsd"
  PolicySetId="MyPolicySet"
  PolicyCombiningAlgId="&xacml;policy-combining-algorithm:deny-overrides">
   <Target />
   <Policy PolicyId="SamplePolicy"
     RuleCombiningAlgId="&xacml;rule-combining-algorithm:permit-overrides">
       <Target>
           <Resources>
               <Resource>
                   <ResourceMatch MatchID="&xacml;function:string-equal">
                       <AttributeValue DataType="&xs;#string">SampleServer</AttributeValue>
                       <ResourceAttributeDesignator DataType="&xs;#string"
                         AttributeId="&xacml;resource:resource-id" />
                   </ResourceMatch>
               </Resource>
           </Resources>
       </Target>
       <Rule RuleId="LoginRule" Effect="Permit">
           <Target>
               <Actions>
                   <Action>
                       <ActionMatch MatchID="&xacml;function:string-equal">
                           <AttributeValue DataType="&xs;#string">login</AttributeValue>
                           <ActionAttributeDesignator DataType="&xs;#string"
                             AttributeId="&xacml;action:action-id" />
                       </ActionMatch>
                   </Action>
               </Actions>
           </Target>
           <Condition>
               <Apply FunctionId="&xacml;function:and">
                   <Apply FunctionId="&xacml;function:time-greater-than-or-equal">
                       <Apply FunctionId="&xacml;function:time-one-and-only">
                           <EnvironmentAttributeDesignator  DataType="&xs;#time"
                             AttributeId="&xacml;environment:current-time" />
                       </Apply>
                       <AttributeValue DataType="&xs;#time">T9H</AttributeValue>
                   </Apply>
                   <Apply FunctionId="&xacml;function:time-less-than-or-equal">
                       <Apply FunctionId="&xacml;function:time-one-and-only">
                           <EnvironmentAttributeDesignator  DataType="&xs;#time"
                             AttributeId="&xacml;environment:current-time" />
                       </Apply>
                       <AttributeValue DataType="&xs;#time">T17H</AttributeValue>
                   </Apply>
               </Apply>
           </Condition>
       </Rule>
   </Policy>
   <Rule RuleId="FinalRule" Effect="Deny">
       <Target />
   </Rule>
</PolicySet>
```

Figure 1: Simple XACML policy generated by the policy editor.

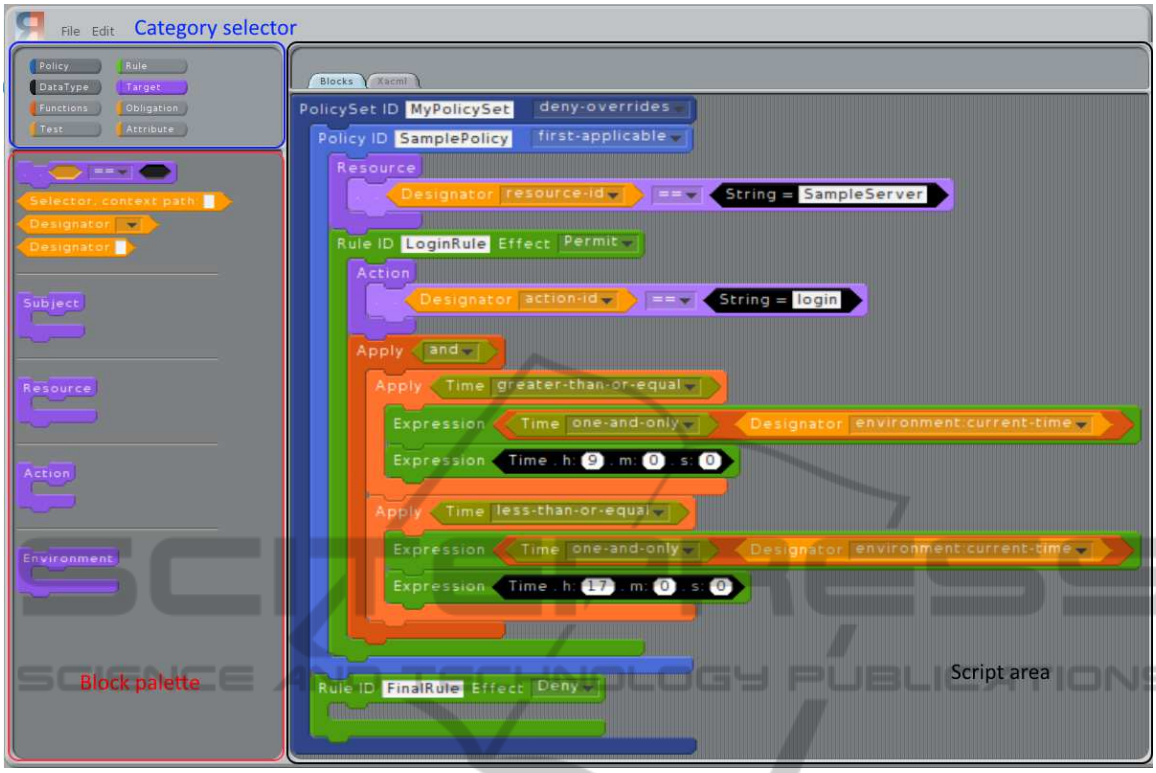

Figure 2: Simple policy using the XACML policy editor based on Scratch.

(Malan and Leitner, 2007). It is a graphical language which defines a set of programming constructs which can be put together as puzzle pieces in order to define a computer program (Malan and Leitner, 2007). The language enforces that only blocks that fit logically together according to the language syntax will stick together.

We believe that a high-level policy language editor for policy makers is needed, to avoid much of the underlying distraction and syntactic complexity of XACML. The two examples below illustrate this. Figure 1 shows all the complexity and intricacies of an XACML policy written in XML. The XACML has been simplified somewhat by denoting the XACML namespace as *&xacml;* and the XML Schema namespace as *&xs;*. This is a simple XACML policy example[1] that applies for requests to a server called *SampleServer*, with a rule that matches a login action and contains an XACML Condition stating that the Subject only is allowed to log in between 09:00 and 17:00.

Figure 2 shows the same policy implemented using our XACML policy editor. The syntactic blocks in the Scratch based XACML editor is able to hide much of the complexity involved in writing XACML statements by providing features such as:

- managing XACML identities and XML schema data types;
- automatically matching attribute designators to the context they are in and the data type they belong to;
- automatically inferring some XML elements, for example the <*Condition*> clause;
- performing run-time type checking operations, ensuring that only sensible XML elements can be put together.

Still the question remains - why write another XACML policy editor, and not reuse and extend one of the existing XACML policy editors? A basic requirement for us is that the policy editor would need to be Open Source, since we want it to be freely available and possible to adapt to a user's specific needs.

One example of an open source policy editor for XACML is the UMU-XACML editor[2]. This editor is made by the University of Murica in Spain. The editor is written in Java and essentially manages a DOM tree with XACML nodes, and provides a user interface with sensible default values or choices for each

[1]The policy example was inspired by http:// www.oasis-open.org/committees/download.php/2713/Brief _Introduction_to_XACML.html

[2]UMU-XACML-Editor: http://umu-xacmleditor.sourceforge.net

type of DOM nodes in the XACML document. The editor supports folding down elements within a given policy in order to view parts of the DOM tree. The editor does not yet support unfolding everything, which makes it cumbersome to get an overview over anything but very small policies. The folding mechanism is problematic from a usability perspective, since the policy-maker does not get an overview over the entire policy.

Another problem with this approach, is that the details of each XACML element is shown in a separate window, which means that it is not possible for a policy-maker to get an overview over how a given policy works without reading the generated XACML. Furthermore, some choices are missing, for example for choosing functions. In total, UMU-XACML does not reduce the overall complexity in writing XACML policies much. UMU-XACML will in other words aid the user in creating an XACML policy, but it has some severe usability issues that makes it undesirable as a design base for our policy editor.

The WSO2 Identity Server[3] is a complete identity management solution that has a web based user interface for designing XACML policies. This interface is from a structural perspective quite similar to UMU XACML, but provides web based forms for generating different policy templates (simple, basic or standard), as well as having a separate policy set editor. This approach has similar deficiencies as the UMU XACML editor since it is difficult to get an overview over the policies without reading the generated XACML. Our approach aims on the other hand at giving the policy maker all necessary information in order to understand the policy in an easily readable high-level graphic language, instead of using a program that creates a forms-based user interface for generating XACML.

Axiomatics has created a freeware policy editor that uses a simplified policy editor language called Axiomatics Language for Authorization (ALFA), which can be used to generate XACML policies (Stepien et al., 2009). This approach aims at achieving similar objectives as our project, by simplifying the policy language used to generate XACML policies. The policy editor is an Eclipse plugin that provides a language that is syntactically similar to Java or C#[4].

Others have also taken a similar approach, by defining user-friendly domain-specific languages for

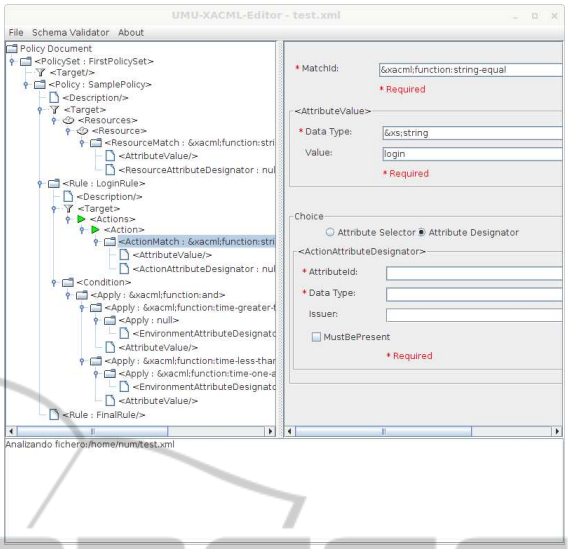

Figure 3: Simple policy using UMU XACML Editor.

implementing parts of the XACML syntax. One such example is easyXACML, which has implemented an XACML editor for the *target* section of XACML and a constraints editor for non-technical users (Stepien et al., 2009; Stepien et al., 2011). This approach is in some ways similar to ALFA and Ponder2 XACML policy integration (Zhao et al., 2008), by defining domain-specific high-level authorisation policy languages. We believe our solution achieves much of the same objective by providing a rich and simple graphical programming environment based on Scratch, which is well known for being easy to use.

Another simplified authorisation language is Ponder2, developed at Imperial College, London which is a general purpose authorisation environment for embedded devices (Twidle et al., 2009). The policies are written in a high-level language called PonderTalk, which is based on Smalltalk. Ponder2 is a powerful environment, but it lacks a high-level graphical language that can aid policy-makers on how to put together policies. PonderTalk therefore has a higher starting threshold for writing policies than our solution, since it requires the policy-makers to learn a subset of Smalltalk as well as how to write the policies in PonderTalk. Another disadvantage is that Ponder2 cannot generate XACML policies, which is required by our use cases (Ulltveit-Moe and Oleshchuk, 2015).

Our solution could in principle be extended to achieve the same benefits as PonderTalk, by supporting a message passing interface (Twidle et al., 2009), since our solution also is based on Smalltalk. This is another observation that went in favour of using Scratch as design base. However adding a message passing interface like this would mean evolving away from the core XACML standard.

[3]WSO2 Identity Server: https://docs.wso2.com/ display/IS450/Creating+an+XACML+ Policy.

[4]Axiomatics Language for Authorization (ALFA) http://www.axiomatics.com/axiomatics-alfa-plugin-for-eclipse.html

A major requirement of the policy editor, in addition to writing general XACML policies, is being able to support writing anonymisation policies for XML documents (Ulltveit-Moe and Oleshchuk, 2015). This means that Ponder2 is not a suitable design base for us.

This overview over different XACML editors shows that there is a need for a good XACML editor that is able to provide a simplified policy development language for policy-makers. All existing environments have their disadvantages with respect to usability and other issues; many DOM-tree-based XML editors require you to edit sub-tree-objects by "zooming in"; clicking on them to open them up and show the details, without allowing the user to see the big picture with all details at the same time. Our solution avoids this problem by providing all information available for the user in the simplified graphical language, so that there is no need to zoom in or out of the policy.

There are many language workbenches and tools for creating editors - both textual and graphical, but from a usability perspective even text editors with language assistance, are too unstructured for XML in general and in particular for such a highly structured language as XACML. On the other hand, creating a diagram-like graphical syntax for XACML using graphical editor tool-kits like GMF[5] would be an option, but diagrams tend to take a lot of space and it is easy to lose the overview in a similar way as with DOM-tree-based editors.

But why choose Scratch of all things - a programming language designed for children? We wanted a highly structured design that was radically different from existing Java/Eclipse based editors.

Other possible design bases exist, for example MIT App Inventor for Android. This is a similar block programming language based on Scratch which can be used for designing mobile apps (Roy, 2012). App Inventor was considered being too tied to the underlying Android operating system and was therefore rejected.

There are also other blocks based programming languages, for example UML (Fowler, 2004), Ladder (Hammond and Davis, 2005) or Lego Mindstorm (Ferrari et al., 2003). However Scratch is considered one of the early models of such languages with the necessary functionality which is open source and was therefore chosen. We did not base the editor on Scratch 2.0, to avoid dependencies to Flash, and ended up using a version of Scratch 1.3 ported to the Pharo Smalltalk engine called Phratch[6]. The Pharo Smalltalk environment[7] was chosen instead of Squeak to get a more modern look and feel on the development environment than the venerable Smalltalk-80 user interface.

3 EDITOR DESIGN

The underlying idea is to use the same approach as Scratch has done with its environment, but using the blocks to express XACML elements. The pure XACML can then be hidden behind the scenes and a simplified representation can be used to create and express policies. Simplifications are done by using blocks for elements and their arguments for the attributes, the policies are then built up by placing and stacking blocks onto each other forming a functional policy. Using this representation of graphical blocks and arguments enables a type-what-you-need based design, letting the user focus on the important policy logic and not the XACML XML syntax. Another useful feature with using graphical elements is that the possibility for user error due to misspellings and syntax errors is reduced. The arguments and blocks can have constraints in them enabling only blocks that are applicable can fit.

The main goal for the editor is to generate correct XACML according to the XACML 2.0 Standard (Moses, 2005). Support for generating XACML 3.0 is left as future work. This implies also that the editor should be able to express the whole or at least the most frequently used parts of the standard.

4 AN EDITOR FOR XACML - IMPLEMENTATION

The editor implementation is based on Phratch, an editor for graphical programming based on Scratch (Malan and Leitner, 2007; Resnick et al., 2009). Phratch uses the Pharo 3.0 environment (Bera and Denker, 2013), which is portable across most common operating systems. An advantage by using Smalltalk, is that this is an agile development environment suitable for rapid prototyping. This means that we quickly can test out ideas and modify the functionality if it does not work out as well as expected. Implementing the editor was done by building upon a subset of Phratch. The visualisation for the blocks and the arguments is the same for the editor as it is

[5]Graphical Modelling Framework, http://www.eclipse.org/modeling/gmp/

[6]Phratch: http://www.phratch.com/

[7]Pharo: http://pharo.org

Figure 4: Block placement indicator.

from Phratch since they represent a similar top down structure as the original XACML would have with its nested elements.

4.1 User Interface

The user interface consist of three parts; a category selector, a block palette, and a script area, as illustrated in Figure 2. The category selector enables the differentiation of blocks into categories to group related blocks. Under the category selector is the blueprint for the different blocks. It contains a block palette that shows the available blocks for the selected category. Each category has a base colour that their respective blocks uses, if not specified otherwise. Blocks shown in the palette can be dragged onto the script area and used in the policy.

When placing a new block onto an existing policy in the script area, the editor will indicate if that block can fit where the user wants it to be. Figure 4 shows what the indication looks like in the editor. If the block could not be placed in the slot due to the rules and constraints given, then no indication for it would be given, and it would be rejected from being placed there.

4.2 Blocks

There are four different base type of blocks implemented into the editor, Figure 5 illustrates their shapes. The first two blocks *Container*, and *Element* are the ones placed directly onto the script area, puzzling them together into a policy. These blocks represent the XACML elements.

The *Input* types are used to represent XACML attributes and are placed into other blocks colour-marked for that attribute type. The *Input string* block is used for long strings that cannot simply be hidden away in the background, for example XPath expressions.

Each block contains a text describing the specific element and can also contain an arbitrary number of arguments, giving the possibility to present both simple and advanced elements with their attributes from the XACML specification.

Figure 5: Blocks.

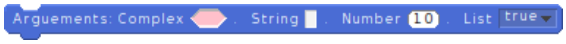

Figure 6: Arguments.

4.3 Arguments

The Arguments used in the editor are illustrated in Figure 6, placed inside an element block, these specify the attributes for the block.

The complex argument with a hexagonal shape, is a place holder for an input block. It is used when more than just one piece of information is needed to be passed along to the main block. Going back to the example from Figure 2, the black input block with the text “String =[ ]” not only returns the string written by the user, but also the data type and its XACML URI representation back to the owner block. The arguments that take a written string can also have the input string block placed into them.

Number arguments are used when the user input is constrained to only integer or floating point numbers.

List arguments act as drop-down boxes for selecting defined items, and can either be a single list or have a multi tiered selection of items.

Arguments can constrain the input blocks or text allowed to be placed into them, for example string and numerical arguments can have constraints in the form of regular expression or a specific interval for numerical values.

To better indicate the allowed *Input* blocks in complex types and where input string blocks should be placed, they are given the same kind of colour as a visual indication. The constraint rules for complex arguments makes it impossible to place the wrong input block into them, reducing user error significantly.

4.4 XACML Text Generation

The XML presentation of XACML is generated on demand when the user wants to export the graphical policy into XACML or preview it using the XACML pane in the editor. Generating the XML starts at the outer topmost block, and recursively calls the continuing blocks in the stack. When all the XML is generated, it runs through a formatting method to enhance the visual appearance.

Each block from the palette has its own unique method for generating its corresponding XML, together with extra behaviour dictated by the shape of the block. An example of this extra behaviour is in

the *container* block which appends the inner blocks, and adds any additional needed elements. This enables the editor to hide elements that do not have any unique attributes, for example the *target* or *condition* elements. Figure 7 shows an example *rule* with an *action* inside it. The generated output is shown in Figure 8, and here the *target* and *actions* elements encapsulates the *action* element. Since these do not contribute to anything but verbosity, the editor adds them automatically when needed. These improvements allow for simplifying the graphic representation of XACML compared to the very verbose XML representation.

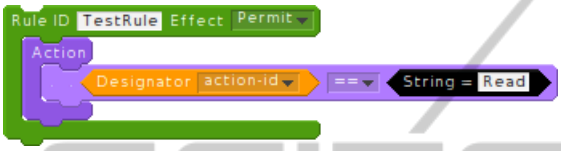

Figure 7: Graphical representation.

```
<Rule RuleId="TestRule" Effect="Permit">
   <Target>
      <Actions>
         <Action>
            <ActionMatch
      MatchID="&xacml;function:string-equal">
            <AttributeValue DataType="&xs;#string">
               Read
            </AttributeValue>
            <ActionAttributeDesignator
               DataType="&xs;#string"
            AttributeId="&xacml;action:action-id" />
               </ActionMatch>
            </Action>
      </Actions>
   </Target>
</Rule>
```

Figure 8: Generated XACML from Figure 7.

takes approximately 26ms, still a relative fast generation time. The second test performed with the hello world example is by nesting the next “hello world” into the “FinaleRule” container block from the previous one. This shows how the generation is over larger inner nesting of blocks, and from Figure 10 it can be seen that the generation has similar generation speed as when blocks are stacked.

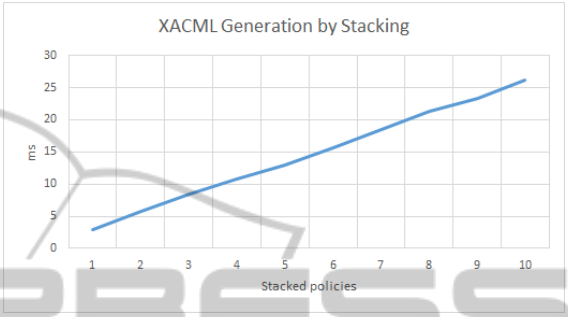

Figure 9: N stacked “Hello-World” policies.

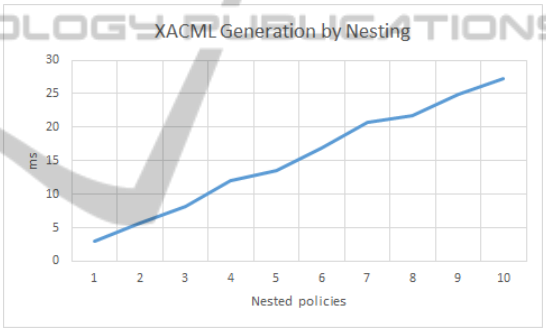

Figure 10: N Nested “Hello-World” policies.

5 DISCUSSION

Our approach has the advantage over existing XACML tools by focusing on the meaning and attributes of the policy, and not the XML syntax of XACML.

Generating XACML from a created policy by the editor is quite fast. Using the policy from Figure 2, tests are done by duplicating this policy and stacking these together simulating a larger policy. The first test illustrates the blocks stacked onto the bottom of the policy under the last rule block. The generation speed for this stacking by up to 10 consecutive times is shown in Figure 9. The time it takes to generate the policy from the “hello world” example is on average 3ms, while stacking 10 of these the generation

The editor now has 80% of elements from XACML 2.0 implemented as blocks. The last 20% are elements used for referring to other policies, version definition and elements used when using self-made policy-combining algorithms. The objective is to implement full support for XACML in the future, however the policy editor is already usable for several simple policies, since the unimplemented elements are less frequently used.

The complex arguments and list arguments ensures that a wrong input cannot be added to them, specific types of number and string literals have constraints that ensures that only a valid input can be typed. The stacking and nesting of blocks does however not yet have all the constraints needed to cover all possibilities. This means that it still may be possible for users not knowing the rules for XACML elements to put together non-functional policies. It is probably not possible to avoid all fault scenarios, however improving the constraints handling is ongoing work where improvements can be added when com-

mon fault scenarios are being detected and mitigated.

Our approach simplifies several XACML concepts significantly, as we have described earlier, which means that overall it should be much easier for policy-makers to write XACML policies than writing such policies using existing tools. User-testing experiments proving that the usability is significantly improved is however left as future work.

In this paper, we let the qualitative arguments for increased usability, based on the heritage from Scratch, speak for themselves.

6 SUMMARY

We have implemented and tested an editor capable of generating XACML 2.0 from a graphical blocks-based representation. this enables the user to focus on the important details and logic of the policy and not on the complex syntax of XACML and XML. It allows policy makers to focus on creating the correct policies, and does not require a significant amount of programming experience to be used. Policies created by the editor are represented in a simplified policy language that is quite close to XACML, and that can be exported to XACML code. The editor uses colour identification and feedback to show the user correct placement of elements into the policy. This reduces the learning curve for implementing XACML policies significantly compared to existing XACML editors, while still keeping core parts of the XACML structure and syntactic elements. Providing a simplified syntax based on Scratch both improves the maintainability and makes it easier to support an agile policy development strategy under dynamically changing environments.

7 FUTURE WORK

Investigating the usability with regards to familiar and non familiar users of XACML is something we would like to do in the future. Users who can write XACML could test the editor by creating large complex policies, and non-technical users creating smaller, simpler policies. The criteria for such a study should include evaluating the time used versus size and complexity of the policy, syntax errors, and logical errors.

Implementing modularisation, and support for multiple active policies in the editor is left for future work.This modularisation will allow the user to break the policies into smaller parts, with clearer intent. these parts can then be reused for the same or other policies as well. Another step beyond modules is to provide the possibility to define and create custom elements and attributes graphical in the editor that then can be used for policies, using the extendible parts supported by XACML.

XACML3.0 support is also left as future work, however we have considered the XACML 3.0 syntax when doing some of the simplifications in the graphic language. Some more work is left on covering the last portion of elements and attributes from the XACML 2.0 standard.

Another advantage by using Phratch as design base, is that this may allow for mixing existing Phratch functionality into the XACML policy generation language in order to define policy templates that can instantiate variations of similar policies easily. This will allow for programmatic generation of authorisation policies, which makes it easier to generate and maintain large and complex authorisation or anonymisation policies. This feature is planned implemented in a future version.

Another possible future improvement, is implementing support for location-based XACML policies based on GeoXACML (Matheus and Herrmann, 2008). This could make it possible to integrate map data, for example from Google maps, into the policy editor, which would allow for defining geographical authorisation constraints on the policies. Further elaboration of such a scenario may be supporting location-based dynamic access control policies for moving objects. This may be useful for designing access control policies for vehicles, boats or other objects moving in a 2D plane, which could be simulated using existing functionality in Scratch. Support for expressing the RBAC profile of XACML (Anderson, 2005), is another possibility for extending the policy editor, for example based on the work in (Ulltveit-Moe and Oleshchuk, 2013; Ulltveit-Moe and Oleshchuk, ; Bonatti et al., 2013).

ACKNOWLEDGEMENTS

This project was sponsored as a summer internship at the University of Agder. The project has also been sponsored by the FP7 EU projects:

PRECYSE - Protection, prevention and reaction to cyberattacks to critical infrastructures, contract number FP7-SEC-2012-1-285181 (http://www.precyse.eu);

SEMIAH - Scalable Energy Management Infrastructure for Aggregation of Households, contract number ICT-2013.6.1-619560 (http://semiah.eu).

REFERENCES

Anderson, A. (2005). Core and hierarchical role based access control (rbac) profile of xacml v2.0. *OASIS Standard*.

Bera, C. and Denker, M. (2013). Towards a flexible Pharo Compiler. In Lagadec, L. and Plantec, A., editors, *IWST*, Annecy, France. ESUG.

Bonatti, P., Galdi, C., and Torres, D. (2013). ERBAC: Event-driven RBAC. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, pages 125–136, New York, NY, USA. ACM.

Ferrari, M., Ferrari, G., Clague, K., Brown, J., and Hempel, R. (2003). *LEGO Mindstorm Masterpieces: Building and Programming Advanced Robots*. Syngress.

Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional.

Hammond, T. and Davis, R. (2005). LADDER, a sketching language for user interface developers. *Computers & Graphics*, 29(4):518–532.

Malan, D. J. and Leitner, H. H. (2007). Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 223–227, New York, NY, USA. ACM.

Matheus, A. and Herrmann, J. (2008). Geospatial extensible access control markup language (geoxacml). *Open Geospatial Consortium Inc*.

Moses, T. (2005). *eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard.*

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: Programming for all. *Commun. ACM*, 52(11):60–67.

Roy, K. (2012). App inventor for android: Report from a summer camp. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 283–288, New York, NY, USA. ACM.

Stepien, B., Felty, A., and Matwin, S. (2009). A non-technical user-oriented display notation for xacml conditions. In Babin, G., Kropf, P., and Weiss, M., editors, *E-Technologies: Innovation in an Open World*, volume 26 of *Lecture Notes in Business Information Processing*, pages 53–64. Springer Berlin Heidelberg.

Stepien, B., Matwin, S., and Felty, A. (2011). Advantages of a non-technical XACML notation in role-based models. In *2011 Ninth Annual International Conference on Privacy, Security and Trust (PST)*, pages 193–200.

Twidle, K., Dulay, N., Lupu, E., and Sloman, M. (2009). Ponder2: A policy system for autonomous pervasive environments. In *Fifth International Conference on Autonomic and Autonomous Systems, 2009. ICAS '09*, pages 330–335.

Ulltveit-Moe, N. and Oleshchuk, V. Mobile security with location-aware role-based access control. In *Security and Privacy in Mobile Information and Communication Systems*, volume 94 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer Berlin Heidelberg.

Ulltveit-Moe, N. and Oleshchuk, V. (2012). Decision-cache based XACML authorisation and anonymisation for XML documents. *Comput. Stand. Interfaces*, 34(6):527–534.

Ulltveit-Moe, N. and Oleshchuk, V. (2013). Enforcing mobile security with location-aware role-based access control. *Security and Communication Networks*, pages 172–183.

Ulltveit-Moe, N. and Oleshchuk, V. (2015). A novel policy-driven reversible anonymisation scheme for xml-based services. *Information Systems*, 48(0):164 – 178.

Zhao, H., Lobo, J., and Bellovin, S. (2008). An algebra for integration and analysis of ponder2 policies. In *IEEE Workshop on Policies for Distributed Systems and Networks, 2008.*, pages 74–77.