ViSPE: A 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
{henrin10,nils.ulltveit-moe,terje.gjosater}@uia.no

Abstract. In this paper we present the Visual Security Policy Editor (ViSPE), a policy-maker-friendly graphical editor for the eXtensible Access Control Markup Language (XACML). The editor is based on the programming language Scratch and implemented in Smalltalk. It uses a graphical block-based syntax for declaring access control polices that simplifies many of the cumbersome and verbose parts of XACML. Using a graphical language allows the editor to aid the policy-maker in building polices by providing visual feedback and by grouping blocks and operators that fit together and also indicating which blocks that stick together. It simplifies building policies while still maintaining the basic structure and logic of XACML.

Keywords: Access control · XACML · Editor · Smalltalk

1 Introduction

The eXtensible Access Control Markup Language (XACML) is a declarative access control policy language, standardised by OASIS, that is implemented in XML [1]. The XACML Standard defines a large set of XML elements and attributes, it 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 does not have 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 part 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 standardization 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,

O. Camp et al. (Eds.): ICISSP 2015, CCIS 576, pp. 107–121, 2015.
DOI: 10.1007/978-3-319-27668-7_7

creating large policies can involve much work. Our solution is to design and develop a user-friendly editor that helps in creating these XACML polices.

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

2 Background and Motivation

The Visual Security Policy Editor (ViSPE) for XACML policies is implemented based on the Scratch programming environment [2,3], on the Pharo Smalltalk engine [4]. It aims at providing a policy-maker-friendly policy description language for designing XACML authorisation policies. It is also a design objective that it shall be able to design XACML-based anonymisation policies for XML documents [5,6]. Scratch is a programming language for children created by the Lifelong Kindergarten research group at Massachusetts Institute of Technology's Media LAB [2,3]. 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 [2]. 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 ViSPE. The syntactic blocks used by the 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 with correct attributes can be put together.

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

```
<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;&algorithm;deny-overrides">
 <Target />
 <Policy PolicyId="SamplePolicy"
  RuleCombiningAlgId="&xacml;&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>
  <Rule RuleId="FinalRule" Effect="Deny">
   <Target />
  </Rule>
 </Policy>
</PolicySet>
```

Fig. 1. Simple XACML policy generated by ViSPE.

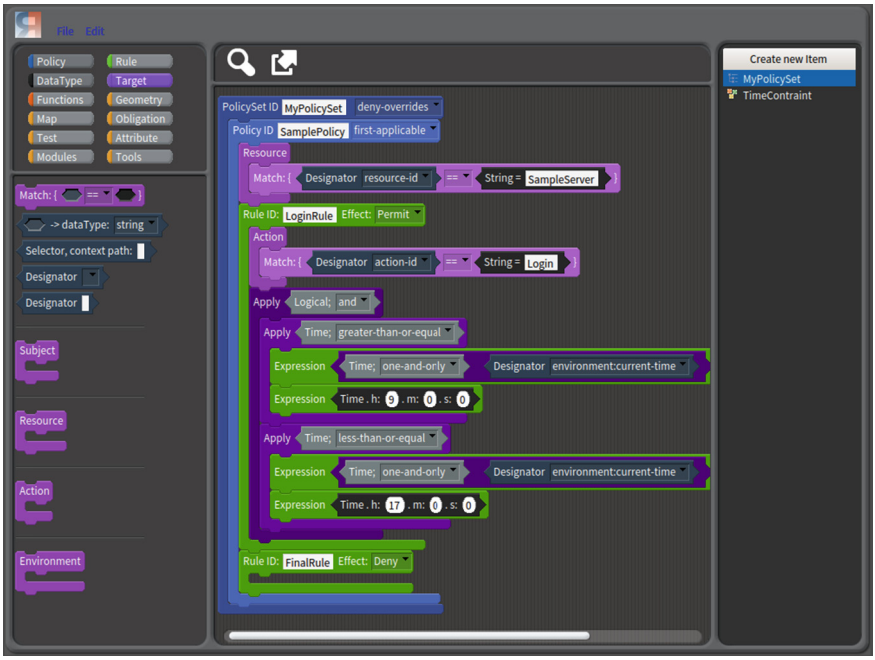

Fig. 2. Simple policy made using ViSPE.

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 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

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

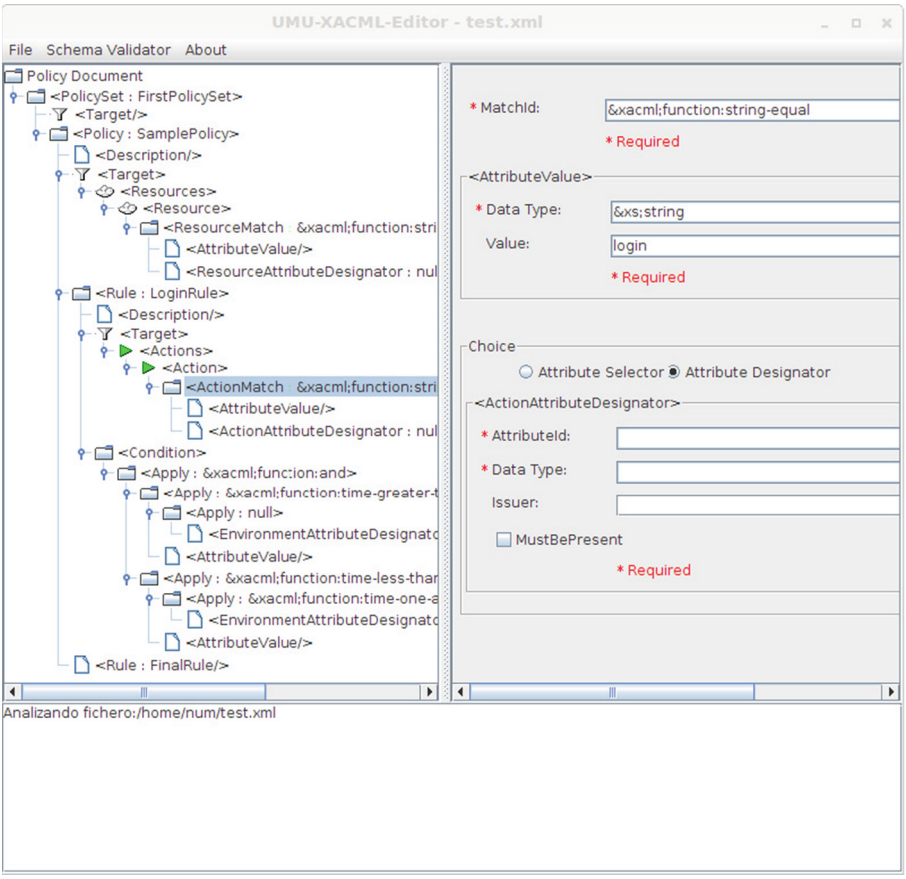

Fig. 3. Simple policy using UMU XACML editor.

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 (Fig. 3).

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.

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

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. 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 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 using their simplified notation aimed for non-technical users [7]. This approach is in some ways similar to ALFA and Ponder2 XACML policy integration [8], 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 [9]. 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 [5].

Our solution could in principle be extended to achieve the same benefits as PonderTalk, by supporting a message passing interface [9], 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.

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 [5]. 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.

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

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 [10]. 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 [11], Ladder [12] or Lego Mindstorm [13]. 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 An Editor for XACML

3.1 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 is 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 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 to fit.

The main goal for the editor is to generate correct XACML according to the XACML 2.0 Standard [1]. 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.

[5] Graphical Modelling Framework, http://www.eclipse.org/modeling/gmp/.
[6] Phratch: http://www.phratch.com/.
[7] Pharo: http://pharo.org.

3.2 Implementation Platform

Our Editor implementation is based on Phratch, an editor for graphical programming based on Scratch [2,3]. Phratch uses the Pharo Smalltalk environment [4], 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 on a subset of Phratch. The shapes used for blocks and arguments is similar for the ViSPE as it is from Phratch, since they represent a similar top down structure as the original XACML would have with its nested elements.

3.3 User Interface

The user interface consists of three core parts; a block palette, a build area, and a management list, as shown from left to right in Fig. 2.

In the top of the block palette is a list of the different categories that blocks are distributed in, while the rest of it presents the available graphical blocks for the category selected. Blocks from the palette can be dragged over to the build area to form the graphical policy.

The management list shows the graphical polices currently active in the editor, and presents the selected policy in the build area. This enables the policy maker to work with multiple polices, and also gives the ability to split them up into different parts which can be useful for large polices with multiple *Policy* and *PolicySet* elements.

In the build area, blocks can be placed anywhere such that a part of the policy can easily be formed before placing it into the policy itself. This also makes it possible to have certain parts that might be used on a later occasion present together with the policy. However only the blocks that are placed into the starting block will be part of the generated policy.

Every new policy starts out with one fixed top level block, used as the starting point for the policy, and can either be a policy-set, or a single policy element.

3.4 Graphical Policy Shapes

For expressing the selection of elements and attributes from XACML, our block syntax uses three different graphical shapes, illustrated in Fig. 4. The first two block shapes are used for expressing XACML elements, while the last one is a

Fig. 4. Example of the different block shapes available.

block for special attributes. Each of these shapes have text embedded in them that is meant to closely resemble their XACML equivalent part.

Attributes for a specific element is also embedded into the block, each attribute has a prefix text combined with an appropriate input field. The most basic of these inputs is a textural or numeric field where the user provides its value. Numerical input fields are differentiated from text fields by providing a more circular shape, illustrated in the *Time* attribute block in Fig. 5. For attributes that only have a fixed number of valid values, an embedded list is provided that presents the user with the allowed choices, and the current selected choice is shown in the block. There is also a special version of the list field that instead of presenting a list of options, opens a dialogue with a custom tailored widget for providing the valid attribute value. An example of such a widget is for the *Date* attribute block shown in Fig. 4, here the list it replaced by a calendar widget, ensuring that only valid dates can be used for this attribute.

For attributes where a simple input is not enough, an attribute block is used. These blocks give the opportunity to add more context to the attribute. An example use for attribute blocks is for expressing the *data-type* attribute, where the block provides the data type as a prefix text, with an appropriate input field for the value itself, such as the *Date* block from Fig. 4. Elements that use such an attribute block has an embedded slot where this can be placed.

Some of the attribute blocks are also used to add more optional information to other attribute blocks. An example of this is the time data type value that might be negative, and also have a Coordinated Universal Time (UTC) component. To express this using the graphical blocks, the *Time* attribute block is placed into a *UTC* block that is placed into a *Negative* block, illustrated in Fig. 5.

Input fields can constrain the input allowed to be placed into them, for example text input fields can use regular expression when accepting new inputs, while numerical fields have the option to only accept values of a certain range.

Some of the list fields can modify the options that are available according to the context they are in, an example of this is the *Designator* attribute block which only shows attribute designators for a specific attribute category if placed inside that category element, otherwise it shows the full selection for all the attribute categories. Placing attribute blocks into their place-holder is also constrained such that only the valid attribute blocks will be allowed to take the place-holders place.

When placing a new block into an existing block in the build area, visual feedback will indicate if that block can fit where the user wants it to be. Figure 6 shows what this visual indication looks like for both element and attribute blocks. If a block cannot be placed in the element or attribute slot due to the rules

Fig. 5. Example of nested attribute blocks.

Fig. 6. Example of the visual placement indicator feedback in action.

and constraints, then no indication is shown, and it will not be placed into the accepting block. Many of the attribute slots is colour coded with the same colour as the attribute blocks that are allowed to be placed into them. For example a black attribute slot means that only attribute blocks in that are in the *DataType* category can be placed there.

3.5 XACML XML Generation

XML can be generated from the graphical policy by using the buttons provided above it. The output can either be shown in a dialogue box that opens inside the editor, or saved directly into a file with name and location chosen by the user.

All the elements that are enclosed by the starting outermost element provided by the policy script are included when generating the policy.

Some of the elements defined in XACML has an outer encapsulation without any attributes. These elements add unnecessary depth and verbosity, but are useful for the XML parsing. Our graphical syntax does not need these outer encapsulations. When generating the XACML XML this outer encapsulation is automatically added for elements that requires it.

An example of this encapsulation is for the different attribute category elements that needs to be encapsulated first by a parent element, and then by a target element. Figure 7 shows the Action element block and the added XACML encapsulation.

During generation, the policy also checks for errors that can occur, and stops the process and notifies the user. The most common error checking is

Fig. 7. Example of automatically added element encapsulation.

for attributes, and it check that text inputs, and slots are not empty. There is also some error checking in the element composition, however there is some more work needed to cover all possibilities. The generated XML is also formatted during generation to give a more readable output, such as adding indenting to each new line according to the current depth of nested elements.

3.6 Additional Features

Some combinations of policy elements might occur often in a policy, but only with slight variations. For example, adding a time constraint for several rules where only the time might be different from each rule. XACML offers some support for dealing with recurring parts using the *VariableDefinition* element where parts of the condition sub-elements can be placed, and later referred to with the use of the *VariableReference* element when both of them has the same identifier. However, this approach only works for parts that are fully static, meaning that it cannot be used for generalising parts that has some variation in them. Our approach to this has been to add support for modularisation. This enables any collection of blocks to be placed in their own script, that can then be merged into a single block usable by the policies. These modules do not only support static declaration of attributes, but also dynamic ones. Providing these dynamic attributes is done by replacing input fields from elements in a module script with place-holders. When all the fields have been defined in a module, it can be merged into one single block where the place-holder variables define the embedded attributes. The newly generated block is then available under the *Module* category of the editor, and can be used as any other block.

An example of how the modules work is presented in Fig. 8. Here, the time constraint from the policy in Fig. 2 is placed into its own module script.

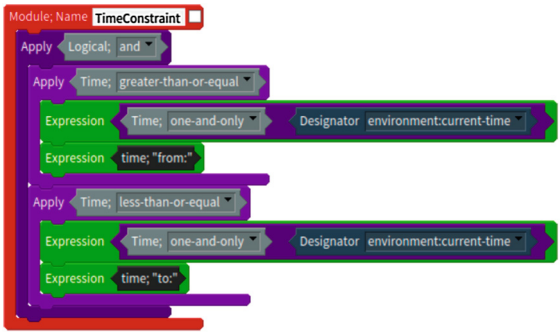

Fig. 8. Modularisation example for a time constraint condition.

The merged block from this module is illustrated in Fig. 9. The embedded attribute slots for this block only accepts an attribute block with the time data type, since this is what the place-holder blocks were defined as. Our module

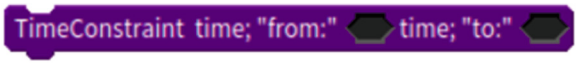

Fig. 9. The merged block made from the module in Fig. 8.

system makes it simple to forge lightweight yet powerful standalone blocks using a subset of the already present blocks.

4 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. ViSPE manages to remove many of the verbose parts from XACML, and bring the focus to the core policy itself.

Generating the graphical XACML policy into XML is quite fast. Using the policy presented in Fig. 2, we have simulated how the generation scales as policies become larger. The simulation was done by duplicating the policy inside the top element, and appending it to the bottom inside of the top element. Each iteration was run 400 times before appending the next policy. The result is presented in Fig. 10, and shows the average time in milliseconds it took to generate N amount of stacked policies.

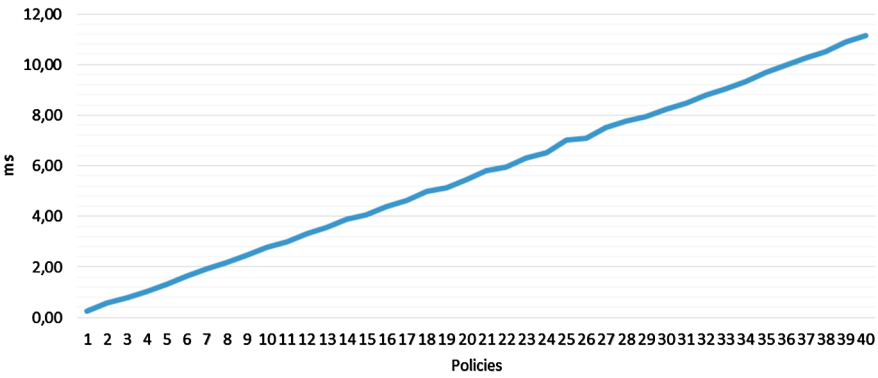

Fig. 10. N stacked policies from the policy presented in Fig. 2.

ViSPE can currently express almost all the elements and attributes from the XACML 2.0 standard. The elements and attributes that are not yet available is the ones used for self-made policy or rule-combining algorithms, and are less frequently used. This means that the editor is fully capable of creating most polices. The attribute slot and list field ensures that a wrong input cannot be added to them. Specific types of number and string literals have constraints that

ensure 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 common 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 create XACML policies with ViSPE than writing such policies using existing tools. An example of this is the policy presented in Figure, here ViSPE uses a total of 31 blocks to generate 3053 characters where only 63 them is typed by the policy maker. 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.

5 Summary

We have designed, implemented, and tested an editor capable of generating XACML 2.0 from a graphical blocks-based representation. Our approach enables the user to focus more on the important details and logic of the policy and not on the complex syntax of XACML, or correct XML formatting. It allows policy makers to focus on creating the correct policies, and does not require a significant amount of programming experience. Policies created by the editor are represented in a simplified policy language that is quite close to XACML, and can be exported to XACML code. The blocks uses colour identification, and visual feedback to show the correct placement of elements and attributes in 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. ViSPE also gives the policy maker the ability to take a subset of the policy and generalise it into one single block. This enables commonly used parts to be abstracted away, presenting a more simplified version that has the flexibility needed for reuse.

6 Future Work

Investigating the usability with regards to users familiar and non-familiar with 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.

XACML 3.0 support is also left as future work, however we have considered this syntax when implementing 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.

Adding additional validation and analytical mechanisms into the editor could be beneficial. Such mechanisms could provide useful information about policies, for example when policies themselves or some of their rules contradict each other. Furthermore, it could be beneficial for the editor to have the ability of also creating the request context schema as graphical blocks, and uses these for testing different request towards a created policy. It would then also be possible to use some of the underlying Scratch framework to visually see the blocks evaluated down into the response.

Another useful feature that could be added is higher level visualizations of the policy, an example of this would be to see a tree structure from the top level policy Set down to specific rules in a policy where each layer shows their concrete attributes, so one can easily get an overview over where different attributes are invoked.

Another possible future improvement, is implementing support for location-based XACML policies based on GeoXACML [14]. This could make it possible to integrate map data, for example from OpenStreetMap, 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 [15], is another possibility for extending the policy editor, for example based on the work in [16–18].

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

1. Moses, T.: eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard (2005)
2. Malan, D.J., Leitner, H.H.: Scratch for budding computer scientists. In: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education. SIGCSE 2007, pp. 223–227, New York, NY, USA. ACM (2007)

3. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: programming for all. Commun. ACM **52**(11), 60–67 (2009)
4. Bera, C., Denker, M.: Towards a flexible Pharo compiler. In: Lagadec, L., Plantec, A. (eds.) IWST. Annecy, France, ESUG (2013)
5. Ulltveit-Moe, N., Oleshchuk, V.: A novel policy-driven reversible anonymisation scheme for xml-based services. Inf. Syst. **48**, 164–178 (2015)
6. Ulltveit-Moe, N., Oleshchuk, V.: Decision-cache based XACML authorisation and anonymisation for XML documents. Comput. Stand. Interfaces **34**(6), 527–534 (2012)
7. Stepien, B., Felty, A., Matwin, S.: A non-technical xacml target editor for dynamic access control systems. In: 2014 International Conference on Collaboration Technologies and Systems (CTS), pp. 150–157. IEEE (2014)
8. Zhao, H., Lobo, J., Bellovin, S.: An algebra for integration and analysis of ponder2 policies. IEEE Workshop Policies Distrib. Syst. Netw. **2008**, 74–77 (2008)
9. Twidle, K., Dulay, N., Lupu, E., Sloman, M.: Ponder2: a policy system for autonomous pervasive environments. In: Fifth International Conference on Autonomic and Autonomous Systems, 2009, ICAS 2009, pp. 330–335 (2009)
10. Roy, K.: App inventor for android: report from a summer camp. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, SIGCSE 2012, pp. 283–288, New York, NY, USA. ACM (2012)
11. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, Boston (2004)
12. Hammond, T., Davis, R.: LADDER, a sketching language for user interface developers. Comput. Graph. **29**(4), 518–532 (2005)
13. Ferrari, M., Ferrari, G., Clague, K., Brown, J., Hempel, R.: LEGO Mindstorm Masterpieces: Building and Programming Advanced Robots. Syngress, Rockland (2003)
14. Matheus, A., Herrmann, J.: Geospatial extensible access control markup language (GeoXACML). Open Geospatial Consortium Inc. (2008)
15. Anderson, A.: Core and hierarchical role based access control (RBAC) profile of XACML v2.0. OASIS Standard (2005)
16. Ulltveit-Moe, N., Oleshchuk, V.: Enforcing mobile security with location-aware role-based access control. Security and Communication Networks, pp. 172–183 (2013)
17. Ulltveit-Moe, N., Oleshchuk, V.: Mobile security with location-aware role-based access control. In: Prasad, R., Farkas, K., Schmidt, A.U., Lioy, A., Russello, G., Luccio, F.L. (eds.) MobiSec 2011. LNICST, vol. 94, pp. 172–183. Springer, Heidelberg (2012)
18. Bonatti, P., Galdi, C., Torres, D.: ERBAC: event-driven RBAC. In: Proceedings of the 18th ACM Symposium on Access Control Models and Technologies, SACMAT 2013, pp. 125–136, New York, NY, USA. ACM (2013)