Access Control Policy Verification and Repairing in Alloy

Alexandr Murashkin, Ming Matthew Ma
The David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{amurashk, m22ma}@uwaterloo.ca

ABSTRACT

Databases can store important information that may be accessible in non-intended way. Therefore, data access control becomes significant [3]. To express access control policies, several languages such as XACML [9] are used to specify whether subjects are allowed to access sets of resources or services to perform specific actions on them.

In this paper, we address the problem of verification and repair of access control policy specification [1] . We develop a tool based on first-order logic modeling to detect and display possible conflicts within sets of access control policies expressed in XACML. We first translate the specification into a relational first order logic language called Alloy [6], and then analyze interactions and conflicts among access control policies using Alloy Analyzer. We then propose potential repairs to the user through the user interface, and automatically apply the fixes selected by the user. We have shown that our tool can automatically determine inconsistencies in user supplied specification, evaluate possible fixes and successfully apply the selected repairs to the specification.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUI)*

General Terms

Computer Aided Verification, Access Control Policy, Verification, Repair

Keywords

access control policy, Alloy, XACML, GUI

[1]We use the term "specification" rather than "model" to avoid ambiguity

1. INTRODUCTION AND MOTIVATION

Although XACML has already achieved a considerable degree of industrial acceptance [8], managing access control policies is still error-prone and complicated [4]. "Nowadays data is increasingly available on-line through the web and other distributed protocols. This heightens the need to carefully control data access" [3]. For example, students should be able to view their marks, but they are not allowed to modify their marks. This is the responsibility of professors, and they have to be able to modify student marks as well as view them. All such rules are defined within access control policies.

However, "access control means not only preventing the leakage of data but also permitting access to necessary information" [3]. Therefore, faults in access control policies that allow students to modify their grades or restrict the access of professors to doing that are both dangerous.

The larger the system, the more complicated access control is required. Moreover, in big systems, access control policies may be defined in multiple locations, they are being refined by different people several times. Therefore, it is hard to keep track of all consequences [3]. This, in turn, makes the error localization hard.

Even if an error is localized, there can be many ways to repair access control policy. Manual repairing without tool support is complex and error-prone. We will justify this problem in the following section and show in this paper that such inconsistencies can be detected and fixed automatically using our tool.

1.1 Problem Statement

The process of verifying and repairing of access control policies is time-consuming, and development of verification tools like Margrave [3] is good evidence of that. Once inconsistencies are found, the user has to think on how to repair policies, but there is no guarantee that the fixes can indeed repair the specification without introducing new inconsistencies. Thus, the entire process of defining, finding inconsistencies and repairing can be overwhelming for the user.

In this paper, we propose a tool we developed that can automatically determine inconsistencies (or faults), recommend the repair to the users and automatically apply fixes to the access control specification for a subset of XACML.

1.2 Paper structure

In the *Background* section we provide relevant information on verification and repair, XACML and Alloy.

All related work we considered is listed in *Related work* section.

Theoretical approaches section will introduce verification and repair approaches we used for our tool.

In the *Implementation* section we will present the components of our tool and its implementation at different stages. *Evaluation* section contains our toy experiment results. In *Future work* section, we discuss our tool limitations and bring up possible future improvements. Conclusion section summarizes completed work and achieved results.

2. BACKGROUND

In this section, we introduce verification and repair, XACML standard for defining access control policies and the Alloy language that we used for verification in our work.

2.1 XACML access control policies

XACML is OASIS standard and stands for eXtensible Access Control Markup Language [9]. It defines structure, interaction and conventions related to policy sets, policies, rules and request within an access control system. In our work we use a simplified subset of XACML used in [8]. In our case, policy set is a set of policies composed using policy combining algorithm, which can be *PermitOverrides*, *DenyOverrides* or *OnlyOneApplicable*[2]. Each policy contains the set of rules and rule combining algorithm which can be *PermitOverrides* and *DenyOverrides*. Each request contains one subject, one action and one resource (Fig. 1), and the system calculates the response of applicable rules, policies, and can make desicions based on that.

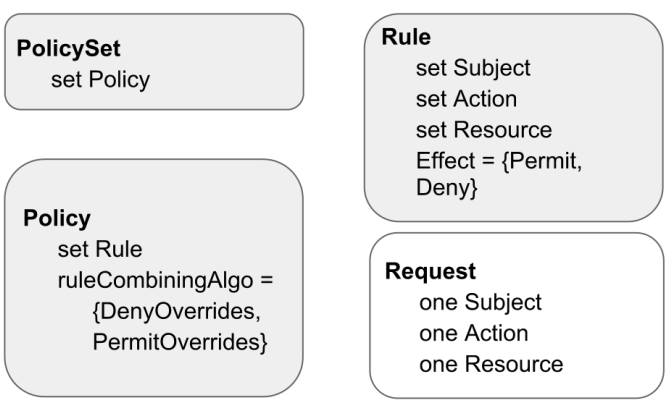

Figure 1: Shortened representation of some XACML entities: policy set, policy, rule, and request. Alloy-friendly notation

A decision can be *Permit*, *Deny* or *NotApplicable*; the latter says that the request is not handled by certain rule, policy or policy set.

Usually XACML access control policies are stored in XML files. For ease of discussion we print XACML policies in a tabular format in this paper, see Fig.2.

PolicySet PS

Policy P1	Policy P3
Rules: 1) Student ⇒ Read ⇒ Marks ⇒ *Permit* 2) Professor ⇒ (Read | Modify) ⇒ Marks ⇒ *Permit*	**Rules:** 1) Professor ⇒ (Read | Modify) ⇒ Marks ⇒ *Deny* 2) Professor ⇒ Modify ⇒ Marks ⇒ *Permit* 3) Professor ⇒ Read ⇒ Marks ⇒ *Permit*
RuleCombiningAlgorithm: DenyOverrides	**RuleCombiningAlgorithm:** PermitOverrides

Figure 2: Valid policy set PS contains two policies: P1 and P3, each has a set of rules and rule combining algorithm. In policy P3, the second and third rules are dominant to the first one because of the rule combining algorithm

So, in our example of a consistent policy set (Fig.2), a policy set PS contains two policies: $P1$ and $P3$, the first rule of $P1$ is $Student \Rightarrow Read \Rightarrow Marks \Rightarrow Permit$ is applied when the subject has the role $Student$, the resource is the $Marks$ file and the action is $Read$, and this access is granted ($Permit$). In policy $P3$, the first rule denies both reading and modifying marks for professor. The policy has *PermitOverrides* rule combining algorithm which says that this rule is dominated by others in case that more than one rule is applicable for some relevant request. The policy response is equal to the response of the rule which is the only applicable, or by the response of dominant rule if multiple rules are applicable. The same is true for policy set response, but it has *OnlyOneApplicable* policy combining algorithm as well, which says that the policy set response in indeterministic if more than one policy is applicable in the context of a given request.

2.2 Verification and repairing

PolicySet PS2

Policy P1	Policy P3
Rules: 1) Student ⇒ Read ⇒ Marks ⇒ *Permit* 2) Professor ⇒ (Read | Modify) ⇒ Marks ⇒ *Permit*	**Rules:** 1) Professor ⇒ (Read | Modify) ⇒ Marks ⇒ *Deny* 2) Professor ⇒ Modify ⇒ Marks ⇒ *Permit* 3) Professor ⇒ Read ⇒ Marks ⇒ *Permit*
RuleCombiningAlgorithm: DenyOverrides	**RuleCombiningAlgorithm:** DenyOverrides

Figure 3: Faulty policy set PS2. In policy P3, the second and third rules are dominated by the first rule, the rule combining algorithm DenyOverrides says "deny is dominant"

Given a specification M, some property φ and satisfaction

[2]we do not consider *FirstApplicable* defined in the XACML standard [9] since it will introduce ordered relations and make our specification more complicated. We also omit dashes and use capitalization for consistency with our specificationl

relation $\models$, verification is the process of determining whether $M \models \varphi$. If $M \not\models \varphi$, the repairing is the process of finding some M', such that $M' \models \varphi$ and M' is derived from M'. This derivation should be minimal, but we need to define this criterion and it depends on the domain we consider.

In this paper, we will try to verify and repair specifications that represent access control policies.

2.3 Alloy

Alloy [6] is a modeling language based on relational first-order logic. We model concepts in Alloy using so called signatures and relations. An abstract signature operates as a set, type or class definition. For example, we can define a policy as follows (as in [8]):

```
abstract sig Policy {
  policyTarget : one Target,
  rules : set Rule,
  combiningAlgo : one RuleCombiningAlgo
}
```

The keyword *extends* can operate like instantiation of a class, and we can define atom (an atomic signature) that "extends" another signature (actually, the former becomes an atomic element of the set of the latter [6]). So, Policy1 becomes an instance of Policy:

```
one sig Policy1 extends Policy {}{
  policyTarget = T0
  rules = Policy1_Rule_Student_Read_Marks_Permit
    + Policy1_Rule_Professor_ReadModify_Marks_Permit
  combiningAlgo = DenyOverrides
}
```

A relation is a set of tuples that relates atoms [6]. In our example, *rules* is relation that relates each policy to a set of rules. Constraints are typically represented in Alloy as global or local facts. A fact is a logical statement that is assumed to always hold. In our example above, "assignments" of $policyTarget$, $rules$ and $combiningAlgo$ are indeed local facts.

We can also specify functions that take in atoms and return other atoms. In our case, we define such functions to calculate policy response with overridden rules. Alloy also supports predicates that return $true$ or $false$ depending on their input parameters.

The Alloy Analyzer is a tool that takes the specification in the Alloy language and finds variables that satisfy them. It can be used both for specification exploration and verification. The way it works in our case we describe in section 4.1.

3. RELATED WORK

Regarding access control policy verification, there are several various related approaches. [5] proposed encoding XACML access control policies ordering relations and then translation of them into SAT solver for verification. This paper defined some Alloy notation as well. The papers [4] and [8] defined XACML access control policies notation in Alloy and deal with verification problem. The latter approach - [8] - is used in our paper, it introduces the usage of predicates for access control policy verification and validation. We found the way the specifications are defined in this paper straight-forward and easy for repair task, so we used this approach and extended it. Other than that, [2] describes inconsistency checking in role-based access control policies (RBAC). [3] proposes Margrave - a tool for access control policies (XACML and other formats) verification and change-impact analysis. [7] used description logics to formalize and verify access control policies. The authors also specified semi-automated access control policy repair as their future work.

None of the paper above solves the problem of access control policy repairing. It does not seem to be published prior related work exactly in this topic and this domain. Rather than that, [11] offers access control policies verification in the language called RW and then synthesize verified specifications in XACML. [1] considers another type of access control policies - XML write access control policies - offers repairs in case of that some actions can be simulated by multiple another actions. Considering different domains, [10] proposes repair trees for inconsistency solving in design models. We use similar trees in our repair approaches. There are works on model repairing in temporal logic, but we do not consider the notion of time in our work. Semi-automated repairing of XACML access control policies is the most important contribution of this paper.

4. THEORETICAL APPROACHES

In this section, we present the verification and repair approaches we use in our work.

4.1 Verification approach

The verification procedure is based on the one in the technical report [8]. We use the same idea in our paper and make it more general. The first step is to define the Alloy input file.

1. First, an abstract specification (or meta-model) is defined at the beginning of the file. This abstract specifcation is domain-specific. It consists of signatures ($Policy$, $PolicySet$, $Rule$, $Effect$, $RuleCombiningAlgorithm$, etc.) and relations ($rules$ relation links a policy and rules it contains).

2. Next, a concrete specifcation is defined. It consists of signatures ($Policy1$, $Policy2$, $Rule1$, $Rule2$) and facts ($Policy1$ contains $Rule1$ and $Rule2$) regarding the relations specified in abstract specifcation. Concrete specification can vary within the given domain, so that one abstract specification can be used for many concrete specifcations within the domain.

3. And finally, a property predicate is defined. This predicate is domain-specific as well, and each predicate represents the negation of the property we need to verify. In this paper, we focused on an abstract property:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

We focused on this property to begin with. Similar property has been specified in [8], but with respect to rules, not policies. However, our approach is not limited to one property: eventually we have to be able to deal with a big set of properties.

The full body of the predicate that corresponds to the property is given in the Appendix A, and its signature looks as follows:

```
pred InconsistentPolicySet [ps : PolicySet,
    req : Request,  p1: Policy, p2: Policy,
    r1: Rule, r2: Rule]
```

The predicate is supposed to find a policy set ps that has $OnlyOneApplicable$ policy combining algorithm and contains policies $p1$ and $p2$, such that policy $p1$ contains rule $r1$, policy $p2$ contains rule $r2$. At the same time, $r1$ defines the response of policy $p1$: the rule $r1$ is either the only applicable rule in context of the given request req, or dominates all other rules of the policy $p1$ after apllying the rule combining algorithm. And similarly, $r2$ defines the response of policy $p2$.

And the second step is to send the Alloy file to Alloy Analyzer and execute the property predicate. We will denote the part of Alloy file without predicate as a specification (so, the specification consists of two parts: abstract specification and concrete one). If the specification is consistent, then the predicate is inconsistent with respect to the specification, and Alloy Analyzer cannot generate an instance (counterexample). If the predicate is consistent, it means that the specification is inconsistent with respect to the property (the property is not satisfied with respect to the specification), and a counterexample is returned.

This verification approach gives certain advantages. We can localize the inconsistency, since Alloy Analyzer will show the values of all the arguments of the predicate in the generated instance, including the request req, even if there is no definition of request. It is not the case if we make the predicate denote the property itself, not its negation: in this case, if the property is not satisfied, then $UNSAT$ cores can be shown, but it takes computation time to minimize them, and the cores still might not be minimal. So, for error localization the approach we took is better.

4.2 Repair approach

In this paper we consider repairing of access control policy specifications. So, there is a policy set PS2 (Fig.3) that is faulty with respect to the same property:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

From PS2, We need to derive a consistent one (there can be many ways of repairing), for example, PS (Fig.2) is one such an option. Our minimality criterion is defined as the smallest number of changes to the faulty policy specification. But the minimality criterion should be explored in the future work.

Once there is an inconsistent policy set, the verification procedure will return the policy set ps, policies $p1$, $p2$ and the corresponding rules $r1$ and $r2$. Our repair procedure is applied to policies or rules, and this will affect the consistency of the policy set ps. Since our verification procedure returns two rules of two policies, actually we can try to apply the same set of repairs to each of them by symmetry. Therefore, we can consider the following repair ways in the context of the returned policy set ps, and one policy $p1$ and rule $r1$ within it.

1. Switch the effect of the rule $r1$. If it was $Permit$, it is changed to $Deny$, and conversely.
2. Switch the policy combining algorithm of the policy $p1$. If it was $PermitOverrides$, it becomes $DenyOverrides$, and conversely.
3. Remove the rule $p1$ from the policy $p1$.
4. Remove the policy $p1$ from the policy set ps.
5. Switch the policy combining algorithm of the policy set ps from $OnlyOneApplicable$ to either $DenyOverrides$ or $PermitOverrides$.
6. Change subjects, actions or resources of the rule $r1$.
7. Add a new rule to the policy $p1$ that will override current responses of other rules.

The ways 1 and 2 seems to be the best. Just switching the values is definitely less radical. Ways 3 and 4 are dangerous: first, they might affect other requests; next, they can bring us to the empty specification which satisfies everything. Way 5 is quite global to apply it, and again, it will hide the conflicts at all, and this can cause more problems in the future. Other ways like changing rule subjects, actions and resources and adding a new rule should be considered in the future, but in this work they were not implemented, because they are potentially risky and may require more efforts to support.

So, for this project, we end up with the first two repair ways and we can apply them to both $p1$ and $p2$, so actually we have 4 primitive repair ways:

1. Switch the effect of the rule $r1$.
2. Switch the policy combining algorithm of the policy $p1$.
3. Switch the effect of the rule $r2$.
4. Switch the policy combining algorithm of the policy $p2$.

However, the application of one repair way might not be enough, in case that the core of inconsistency is not represented by two rules only. First, there might be redundant rules. For example, if the rule $Student \Rightarrow (Read|Modify) \Rightarrow Marks \Rightarrow Permit$ is present, then the rule $Student \Rightarrow Read \Rightarrow Marks \Rightarrow Permit$ is redundant, and changing the

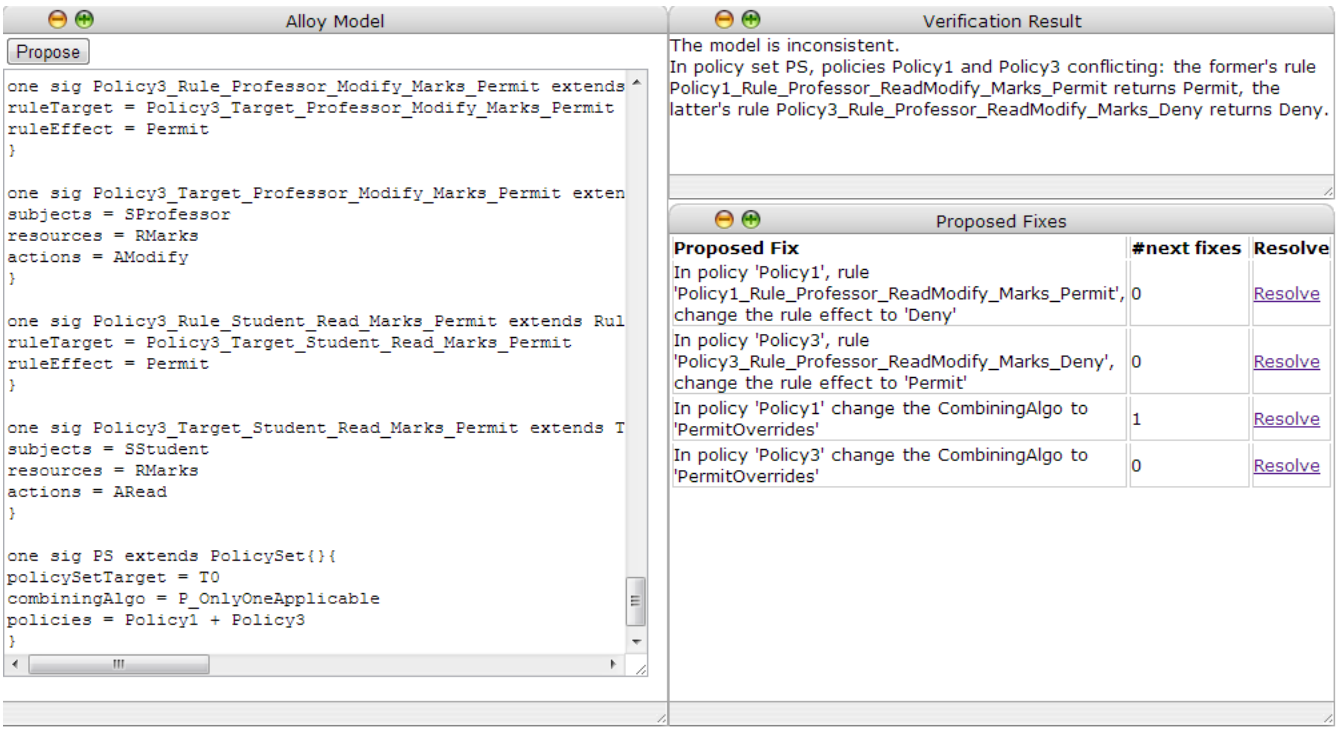

Figure 4: Our tool: the User Interface

rule effect of the latter will cause the application of the former rule and this will not solve the problem in one step. This case is handled during the optimization during the conversion of the source file to Alloy file. Next, there are cases that a policy set is faulty for more than one request (not only for the request $Professor \Rightarrow Read \Rightarrow Marks$, but with the request $Professor \Rightarrow Modify \Rightarrow Marks$, for instance), and it definitely needs more than one repair in our approach. This is why we propose the displaying of the approximate numbers of next fixes.

The approximate number of next fixes is calculated as follows. After repairs have been proposed, the system tries to apply each of them gaining partially repaired specifications. Then, the system runs the verification again for each of the partially repaired specifications. If some partially repaired specification is consistent, it means that this specification is fully repaired, and no more repairs needed. So in this case, the number of next fixes is zero. If some partially repaired specification is still inconsistent, then the system tries to apply fixes to the partially repaired specification again, and adds 1 to the number of next fixes. This process can run infinitely and cause state explosion problem, so we limit the depth to certain amount (in our case we specified this depth to be equal to 2).

If the number of next fixes for each fix is greater than zero, then user needs to apply fixes multiple times. The user will go down the repair tree until he will get the fully repaired specification (Fig.5).

In this approach, more than one path can produce the repaired specification. It does not matter whether this is a

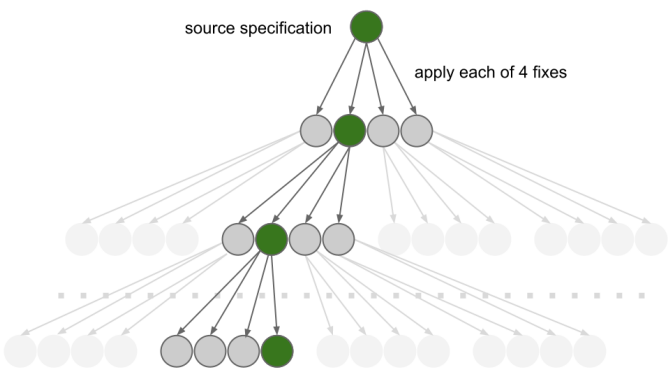

Figure 5: Repair trees. Green nodes are decisions of the user. Grey nodes are partially repaired specifications: dark ones are displayed to the user, light ones are precomputed, but not displayed

minimal one or not: if the user chooses the path with minimal number of next fixes, then eventually he will get the repaired specification with minimal number of changes.

However, there is a problem with this approach - cycles. For example, switching rule combining algorithm may not be helpful. So the system will propose to switch rule combining algorithm and then, at the next step, switch it back. However, our approach helps to solve this problem: if cycles are present, then the way that leads to cycles will be annotated with bigger number of next fixes automatically, so it is less likely that the user will want to choose this way. Still, we can remember last states and detect cycles in another

way.

5. IMPLEMENTATION

The function of our tool is to automatically determine inconsistencies within access control policy defined by XACML, and recommend the potential repair to the user, and finally apply the user chosen fixes automatically. Throughout this process, user only have to review the recommended fixes, and choose the fix by clicking a button in user interface, and then the specification should be fixed automatically by our tool (Fig. 4).

To increase the usability of our tool, we have implemented external module, a converter that can automatically generate Alloy file based on a table template (or XML, CSV file template) so that the user does not need to know Alloy modeling language. As the content of the such a template, the user is asked to specify the rules, subjects, actions, resources and effects of each rule in predefined format. Our converter will than generate the Alloy file. Once Alloy specification is generated, we input the specification into our inconsistency detection and repair tool.

5.1 Architecture and data flow

Our tool consists of three major parts: the User Interface (UI), PropertyVerifier&Fixer and Alloy Analyzer. The architecture is shown in the figure below (Fig.6).

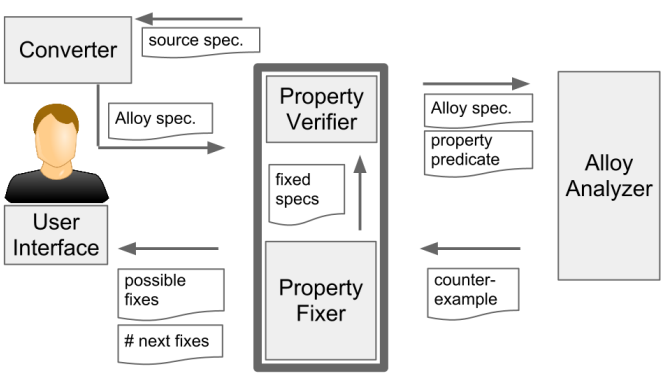

Figure 6: Tool Architecture and Data Flow

At the beginning, the UI first takes in the Alloy specification created, and sends it to PropertyVerifier&Fixer. Next, the PropertyVerifier part starts Alloy Analyzer to processes Alloy specification and its property predicate. It has be noted that we write our predicate in a way that it checks for if two policies return inconsistent results. PropertyFixer then takes in the results from Alloy Analyzer and extracts relevant information, such as inconsistent rules, subject, action, resource and effect, and processes that information to propose the potential fixes. The fix recommendation from PropertyFixer is passed to the UI, and the user is shown with a list of fixes. Once the user chooses one particular fix, our tool will automatically update the original Alloy specification. This updated specification is then passed to PropertyVerifier to check if there are any inconsistencies. This process is repeated until the specification is verified to be consistent.

In order to realize the above mentioned architecture, we have created several Java modules which are summarized below.

5.1.1 InteractiveRepairer

This is the core of our tool, it contains:

- Main engine
- PropertyVerifier
- PropertyFixer

The Main engine controls the communication between the user interface and our Fixer and Verifier; it processes input from UI and outputs the results to UI for user inspection. PropertyVerifier calls Alloy Analyzer through our API and obtains XML representation of the generated instance that contains information about the policies, rules, request and effects. PropertyVerifier then follows XPath standard to extract and store that information from XML for later processing. Once the XML nodes are extracted with relevant information from Alloy Analyzer, PropertyFixer takes in those information and proposes potential fixes. Once the user selects the preferred fix, this decision is communicated to PropertyFixer again through Main engine, and PropertyFixer will automatically apply the fixes to the original Alloy specification.

Note that PropertyFixer and PropertyVerifier function as a pair, and one pair is required for each property to be verified; in our study, we limit the property verification to the one property but this is extensible by creating another pair of PropertyFixer and Verifier.

5.1.2 AlloyRunner

AlloyRunner is an API that facilitate the communication between our tool with Alloy Analyzer. Mainly, it initializes the Alloy Analyzer, sends the specification to Alloy Analyzer, and gets XML instances from it. Alloy Analyzer serves as verification engine for our input specification and will return instances or counterexample which we use in later processing.

5.1.3 Presentation Layer

Presentation layer uses dynamic web application technologies such as Javascript and Node.js to allow interactive specification repair. Node.js executes interactive repair with user inputs and calls back to the Main engine in InteractiveRepairer to apply the repair. For instance, the Propose button triggers Main engine to start PropertyVerifier to execute the verification, and the Resolve links will cause the PropertyFixer to apply the repair on the original Alloy specification. The return values are associated with each function call-back of the button, and represents error location and proposed fixes for Propose command and new specification for Resolve command. All proposed fixes are shown and annotated by the approximated number of next fixes.

6. EVALUATION

Despite XACML is well-adopted standard, we have not evaluated our tool on industry specifications given the time constraints. By now, for our dataset, we have defined 6 toy examples in tabular format. We intentionally made 5 inconsistent examples of policy sets and a consistent one. Policy sets contained from three to four policies with three to six rules each. The tool has successfully identified inconsistent policy sets and proposed the ways of fixing them. The number of next fixes also matched the number we calculated manually. Conducting a proper study with the industry examples is a part of the future work.

7. FUTURE WORK

Although our tool demonstrate potential value in automated access control repair, while significantly reducing user's manual work, we see future improvements in our tool. We noticed the following limitations in our tool:

1. Verification is bounded
2. Can verify one property in a time until first counterexample is found
3. Subset of XACML is covered
4. Repair procedure depends on the property and requires predefined prospective repair ways

The first two points are pretty clear. Since we use Alloy as it is for bounded verification, we can guarantee that the tool will give correct results within the given scope. Although we significantly dependent on Alloy, we mainly interfaced the communication with Alloy API, so in the future, there will not be problems if Alloy back-end changes.

Although we strive to make our tool scalable, the automated Alloy file generation from a user specified table brings limitations in the structure of the specification we generate. Although this access control specification works well for our present study, it only represents a subset of XACML. In the future, we would like to explore different ways to increase the flexibility of our specification. One potential way is to incorporate sketching techniques and having multiple rule tables so that the user can choose the structure of the specification our tool generates.

Even if the state explosion problem has been handled, it is not addressed in the smartest way due to time constraints. By now, we use repair trees and limit the depth of the potential solution to two. This might still bring problems if the specification is large and the number of fixes is big. We would like to study more on ways to deal better with this common problem in verification and repair.

One important future improvement is on the repair recommendation. Currently, we only propose repairs based on rule effect and combining algorithm. We derived these approach from observation and testing. We will explore other repair ways like changing rule subjects, actions and resources, adding new rules and others to enhance our repair framework. We will also explore possible minimal repair definitions in the domain of access control policies.

And finally, we will conduct more effective evaluation of our tool with industry examples.

8. CONCLUSION

In this report, we presented our tool which can automatically determine inconsistencies in XACML access control policies, recommend possible fixes to the user and automatically apply the repair to the specification. We reduced the manual work, and the tool does not require user knowledge of Alloy as a modeling and verification tool.

We have improved the verification approach and abstract specification presented in [8] and proposed repair approach related to this verification approach.

We have implemented the core part of the tool that takes in the Alloy specification and runs Alloy Analyzer in the back-end, and retrieves the information on generated inconsistent instances to propose potential fixes. Among the fixes, it also calculates the approximate number of next fixes required for the selected fix.

We have designed a user interface of the tool that allows user to input a specification or action and displays verification and repair results.

From evaluation results, we have shown the feasibility of our approach and that our tool can help the user to repair inconsistent access control policy specifications effectively, requiring only user input for fix selection as a manual part, the rest actions are automated.

And finally, we have created the input converter so that user can convert policy from other formats.

8.1 Implications

First, we have concluded that semi-automated repair is possible for access control policy defined by XACML structure. We have made implications on the fact that user input is still necessary to reduce the complicated computation and steps taken to fixing the specification. By allowing user to select from the list of recommended fixes, we were able to efficiently repair the specification successfully dealing with state explosion.

Next, we have shown that our tool should be easily extensible so that specification repairing is not limited to the property we considered and access control policies domain we addressed. We see our InteractiveFixer module as a "head" which we can change according to the specification from another domain or other properties that need to be verified.

9. REFERENCES

[1] L. Bravo, J. Cheney, and I. Fundulaki. Repairing inconsistent XML write-access control policies. In *Proceedings of the 11th International Conference on Database Programming Languages*, DBPL'07, pages 97–111, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] V. Cridlig, R. State, and O. Festor. A model for checking consistency in access control policies for network management. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium*, pages 11 –19, 2007.

[3] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 196–205, New York, NY, USA, 2005. ACM.

[4] G. Hughes and T. Bultan. Automated verifcation of access control policies. Technical report, University of California, 2008.

[5] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *Int. J. Softw. Tools Technol. Transf.*, 10(6):503–520, Oct. 2008.

[6] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, revised edition, 2012.

[7] V. Kolovski and J. Hendler. XACML policy analysis using description logics. Technical report, University of Maryland-College Park, Rensselaer Polytechnic Institute, 2008.

[8] M. Mankai and L. Logrippo. Access control policies: Modeling and validation. Technical report, University of Ottawa, 2008.

[9] O. Open. *eXtensible Access Control Markup Language (XACML)*. The MIT Press, 2.0 edition, 2005.

[10] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 220–229, New York, NY, USA, 2012. ACM.

[11] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 56–65, New York, NY, USA, 2004. ACM.

APPENDIX

A. VERIFICATION PROPERTY PREDICATE

The predicate below (written in Alloy) is used for automated verification of the following property:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

```
pred InconsistentPolicySet [ps : PolicySet,
    req : Request,  p1: Policy, p2: Policy,
    r1: Rule, r2: Rule]{

  ps.combiningAlgo = P_OnlyOneApplicable
  p1 in ps.policies
  p2 in ps.policies
  p1 != p2
  r1 in p1.rules
  r2 in p2.rules
  policyResponse[p1, req] = Permit
  (
    p1.combiningAlgo = DenyOverrides and
    (no r1':Rule | r1' in p1.rules
      and ruleResponse[r1', req] = Deny)
    and ruleResponse[r1, req] = Permit
  )
  or
  (
    p1.combiningAlgo = PermitOverrides
    and ruleResponse[r1, req] = Permit
  )
  policyResponse[p2, req] = Deny
  (
    p2.combiningAlgo = PermitOverrides and
    (no r2':Rule | r2' in p2.rules
      and ruleResponse[r2', req] = Permit)
    and ruleResponse[r2, req] = Deny
  )
  or
  (
    p2.combiningAlgo = DenyOverrides
    and ruleResponse[r2, req] = Deny
  )

}
```