Stateful Next Generation Access Control for Fire Response Control

Alperen Tercan
Computer Science Department
Colorado State University
Fort Collins, Colorado
alperen.tercan@colostate.edu

***Abstract*—Many real-world problems require flexible, scalable, and fine-grained access control policies. Next Generation Access Control(NGAC) framework inherits these traits from Attribute Based Access Control(ABAC) and provides an intuitive graph-based approach. In this work, we augment NGAC framework with multi-level rule hierarchy and stateful policies. Then, we show how an NGAC policy can be analyzed together with an environment model using Alloy. This allows defining complex dynamic systems and keeping policies still tractable for automated analysis. We present our approach on emergency fire response problem.**

I. Introduction

The vision for Next Generation Access Control(NGAC)[1] combines previous access models' strengths like flexibility, auditability, and allows high level of granularity. In this work, we extend the NGAC framework to support stateful policies and rule priorities which are desired in certain settings. Stateful policies allow policies to be dependent on a global state variable. Rule priorities extends the two level hierarchy of allowance and prohibition rules to a multi-level hierarchy to provide a more flexible control of exceptions. While some of the dynamic and stateful aspects can be simulated with the *obligations* component of an NGAC policy, the complete permissiveness of obligation operations in the NGAC vision presents a hindrance both to policy comprehension and to formal policy analysis. In our work, we work in an obligation operations free environment; augmenting NGAC with stateful policies and rules priorities in order to express the dynamism of practical policies, while ensuring policy comprehension and policy analysis tractability.

NGAC relations are often represented as directed acyclic graphs(DAGs), so access decisions are computed through efficient graph traversal algorithms. Similarly, many types of queries about the policy can be reduced to known graph problems. While such an approach may allow the use of very efficient algorithms, it carries the downside of the requirement of a graph problem being correctly formulated, and is cumbersome during the policy development process.

A more flexible approach is to use a declarative analysis engine to succinctly specify and answer complex queries for policy analysis. While this approach may be slower, it can support many different query types with minimal overhead to the administrator. Also, since such complex queries are usually needed for offline policy analysis, an efficiency-flexibility tradeoff can be preferable.

Motivated by this observation, we show how our extended NGAC policies can be encoded in Alloy [2]. Then, we show how the analysis engine for Alloy can be used to process and answer complex queries for policy analysis. Finally, we demonstrate the effectiveness of this approach on a sample policy defined for a fire emergency response problem.

In this work, we will firstly discuss some related work and introduce some concepts that we will use in section II and section III. Then, we will define our NGAC framework in section IV and show how it can be coupled with environment model in section V. Finally, we will describe our fire response problem in section VI and discuss our Alloy implementation in section VII.

II. Related Work

Related work to this paper can be divided into two groups with regard to the aspects shared with this work: Using Alloy for policy analysis and applying NGAC to real world problems.

[3] uses Alloy to detect inconsistencies in both eXtensible Access Control Markup Language(XACML) policies and their novel process-based access control work. [4] uses Alloy to analyze integration of multiple access control policies and illustrates its capabilities on a hospital management example. More recent examples of using Alloy for access policy analysis can be seen in [5] and [6]. The first work uses Alloy to analyze an Attribute-Based Access Control(ABAC) policy for Online Social Networks. The second one analyzes a novel Role-based Access Control(RBAC) variant named Emergency-RBAC(E-RBAC) which incorporates Break the Glass policies for emergency response.

On the other hand, there are works that uses NGAC to solve specific access control problems.[7] uses NGAC for implementing an Authentication, Authorization and Accounting(AAA) system for industrial IOT systems. [8] focuses on securing home IOT environments. [9] uses NGAC to secure Mobile Health applications by protecting sensitive healthcare data from unauthorized access. Unlike our work, they use graph database approach for scalable real-time access control queries.

III. Background

In this section, we will summarize NGAC, State Machine, and Alloy to provide the necessary background.

A. NGAC

NGAC is an ABAC framework developed by NIST. It is still more of a vision, as there is not an accepted standard formulation of it. However, there exists some commonly used formulations like [1].

NGAC uses subject and object attributes in order to define fine-grained rules and still maintain the system scalable. Subjects and objects are "assigned" to attributes and access control rules are defined over these attributes. Attributes form a directed acyclic graph(DAG) which creates a hierarchy of attributes. This allows definition of rules with different granularity which gives both fine-grained and scalable policies. Access is allowed if there is a rule allowing the access and there isn't a prohibiting rule. In other words, prohibition rules take precedence.

In this work, we use a more graph and set theory oriented notation. Assign relations are denoted as edges in attribute graphs while Association relations are represented as tuples.

B. State Machine

A finite-state machine(FSM) is a mathematical model of computation.[10] It has a finite set of possible states, hence finite-state. It starts in an *initial state* and it is in exactly one state at any time. Changes between states are called *transitions* and can be dependent on *input* at that time. Transitions can be deterministic or non-deterministic.

A deterministic FSM, M, can be formally defined by a tuple $M = (\Sigma, Q, q_0, \delta, F)$ where:

- Σ is the input alphabet
- Q is a finite non-empty set of states
- $q_0 \in Q$ is a initial state
- $\delta : Q \times \Sigma \rightarrow Q$ is a state-transition function. Hence, current state and input determines the new state.
- $F \subseteq Q$ is the set of final states.

In a non-deterministic FSM, transition function is $\delta : Q \times \Sigma \rightarrow 2^Q$, ie. it returns a set of states. The state machine still can have exactly one state at any time, the returned set of states shows the options. It can take any of the states in the returned set; hence, it is non-deterministic.

We say that $M = (\Sigma, Q, q_0, \delta, F)$ accepts a string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$, if and only if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

- $r_0 = q_0$,
- $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$,
- $r_n \in F$

Then, we say that M recognizes a language A if $A = \{w \,|M \text{ accepts } w\}$.

C. Alloy

Alloy is language for describing structures and a tool for exploring them.[2]. It has been used in a wide range of applications from security mechanism analysis to designing telephone switching networks.

The main intuition behind Alloy's approach to analyzing models and checking assertions is that most of the interesting and potentially violating examples are relatively small ones. Thus, there is usually no need to have the capability to exhausting an entire state space.

Motivated with this intuition, Alloy checks samples of a small scope of the given model to find satisfying or violating instances. Alloy can be used to generate both satisfying examples for a given set of constraints and violating counter-examples for an assertion on a model. This is done by compiling Alloy specifications into a SAT-formulae and using known SAT-solvers to solve them. Once the examples are found, they can be visualized as relation graphs which gives useful insights into the designed model and constraints.

Alloy language uses sets, called *signatures*, to describe a class of objects. These signatures can be organized in order to model hierarchical relations between the classes and it can model many different type of inheritance.

Moreover, *relations* can be defined over signatures to describe the interactions between classes. In addition to supporting object-oriented-like properties to describe compositional relations, one can explicitly define relations over many signatures. Such relations can be considered as sets of n-tuples where elements of tuples come from the defined signatures.

Then, one can write *predicates* over the described signatures and relations and check whether there exist examples in a given scope that violates or satisfies these predicates.

Scope has a specific definition in Alloy. It is determined as how many elements of the top-level signatures can have. Alloy allows setting a global limit that affects all top-level signatures and setting some of the signatures separately as exceptions to the general scope.

IV. Next Generation Access Control

In this section, our NGAC model will be discussed in terms of policy elements, decision functions, and their algorithmic equivalents.

A. Policy Elements

In our work, NGAC policy class has the following elements:

- A set S of *subjects*, a set O of *objects*, and a set Act of actions.
- A finite set Q of environment *states*. Access decisions are allowed to be state dependent. This increases expressive power of the policies. Our NGAC framework does not put any constraints over how states relate to each other and does not make use of such relations. However, modeling state transitions allows analysis of a dynamic system rather than viewing the system as a mere collection of disconnected snapshots. In the next section, we will show how stateful policies and such a model can be combined.
- A set of *subject attributes* A_S, and a set A_O of *object attributes*.
- A set of allowance rules, $Allow$, and a set of prohibition rules, $Prohibit$, where $Allow, Prohibit \subseteq A_S \times 2^{Act} \times \times 2^Q \times A_O$. These two sets of rules specify which action sets are allowed between which subject and which object attributes in which states.

- A priority function $p_k : Allow \cup Prohibit \rightarrow P_k$ where $P_k = \{1, 2, \ldots, k\}$. Hence, p maps allowance and prohibition rules to a set of k priority levels. This k should be chosen according to the application in hand.
- *Acyclic Attribute hierarchy graphs* We have two graphs, the attribute hierarchy graphs for subjects, and for objects which specify the hierarchical relationships between the attributes. We will describe the attribute hierarchy subject graph G_S, the graph $G_O = (V_O, E_O)$ is similar. The graph $G_S = (V_S, E_S)$ is a directed acyclic graph (DAG) as follows.
  - The nodes V_S consist of all subjects and subject attributes, that is $V_S = S \cup A_S$. Notice
  - The edges in E_S specify the attribute relationships to other attributes or to subjects. All the subject nodes $s \in S$ have in-degree 0 (no incoming edges). An edge $(s, a) \in E_S$ where s is a subject and a is an attribute indicates the subject s *has* the attribute a. An edge $(a_1, a_2) \in E_S$ where both a_1 and a_2 are attributes indicates that any subject that has attribute a_1 also has attribute a_2 (in the other direction, it could happen that a subject has attribute a_2, but not a_1). It can be checked that if a subject has an attribute a, and if there is a directed path from a to a' in G_S, then the subject also has attribute a'.

B. Policy Decision Function

We say an action $z_{act} \in Act$ can be performed by a subject s on object o in an environment state q if and only if:

$$\max\{p((a_s, Z_{act}, Z_Q, a_o)) | (a_s, Z_{act}, Z_Q, a_o) \in Allow, \quad (1)$$
$$s \rightsquigarrow a_s, o \rightsquigarrow a_o, z_{act} \in Z_{act}, q \in Z_q\}$$
$$>$$
$$\max\{p((a_s, Z_{act}, Z_Q, a_o)) | (a_s, Z_{act}, Z_Q, a_o) \in Prohibit,$$
$$s \rightsquigarrow a_s, o \rightsquigarrow a_o, z_{act} \in Z_{act}, q \in Z_q\} \quad (2)$$

where $s \rightsquigarrow a_s$ means there is a path from s to a_s in the attribute graph.

C. Answering Access Queries

In this section, we will propose a naive solution to Answering Access Queries, just to give a rough upper bound on the time complexity added by having priorities.

Formally, we want to answer whether subject s can perform z_{act} on object o in environment state q. The conditions for this are stated above and they require finding the highest priority allowance and prohibition rules.

We can reduce this problem into a shortest path problem as below:

Firstly, build graphs $G_A = (V_S \cup V_O, E_A)$ and $G_P = (V_S \cup V_O, E_P)$ where

$$(u, v) \in E_A \iff (u, v) \in E_S \ \vee \quad (3)$$
$$(v, u) \in E_O \ \vee$$
$$(\exists!(u, Z_{act}, Z_Q, v) \in Allow \text{ st.}$$
$$z_{act} \in Z_{act} \wedge z_Q \in Z_Q)$$

and

$$(u, v) \in E_P \iff (u, v) \in E_S \ \vee \quad (4)$$
$$(v, u) \in E_O \ \vee$$
$$(\exists!(u, Z_{act}, Z_Q, v) \in Prohibit \text{ st.}$$
$$z_{act} \in Z_{act} \wedge z_Q \in Z_Q)$$

Intuitively, we connect G_S with G_O by using $Allow$ and $Prohibit$ rules after reversing all edges in G_O. Then, we can reduce the no priorities case in (Prabhu et. al.) to o being reachable from s in G_A but not G_P. So, we can check whether there is path from s to o.

When we have rules of different priorities, the problem becomes a longest/shortest path problem. Assume G_A and G_P has a cost function c such that:

$$c(u, v) = \begin{cases} 0, & \text{if } (u, v) \in E_S \cup E_O \\ 1/p(u, Z_{act}, Z_Q, v), & \text{otherwise} \end{cases} \quad (5)$$

Note that $p(u, Z_{act}, Z_q, v) > 0 \implies c(u, v) \geq 0$. Now, the cost of the shortest path from s to o will be $1/\max\{p((a_s, Z_{act}, Z_Q, a_o))\}$. Then, we can answer queries using any shortest path algorithm.

For example, a standard Dijkstra implementation runs in $O(V^2)$ time. As mentioned before, no-priorities case is a reachability problem. This can be solved by DFS/BFS algorithms which has a time complexity $O(V + E)$.

V. Policy Controlled State Machine

By combining the policy with a model of the environment, we can capture the dynamic nature of a system and analyze its progress from an initial state under the policy. In order illustrate this, we'll assume a finite-state machine as the model for the environment.

Following the formal definition in subsection III-B, we can define the environment model $M = (\Sigma, Q, q_0, \delta, F)$ as follows:

- The alphabet is $\Sigma = S \times Act \times O \cup \{\perp\}$, where $S \times Act \times O$ captures user operations on the system and $\perp$ is no-operation symbol which captures environment changes independent of user activity.
- The set of states is Q, which is also the set of states in the policy. In some cases, many states of finite state machine might be the same for access control decisions. However, we can consider such cases as special cases of this general framework, where significant state aliasing happen.
- $q_0 \in Q$ is an environment initial state.
- $F \subseteq Q$. The set of final states will be dependent on the problem. It is possible that $F = Q$, meaning that there are no failing states.

Transition function is where we will combine the policy with environment model. The policy will prohibit some transitions of the underlying environment model. There are two approaches to implement this.

Firstly, it is conventional to allow δ to be a partial function, i.e. it is not defined for all tuples in $Q \times \Sigma$. If a symbol x is

observed in a state q but $\delta(q, x)$ is not defined, M will reject it. Then, assuming an underlying transition function $\delta_U : Q \times \Sigma$:

$$\delta(q,x) = \begin{cases} \delta_U(q,\perp), & x = \perp \\ \delta_U(q,x), & x \neq \perp \text{ and policy allows } (x,q) \\ \text{undefined}, & \text{otherwise} \end{cases} \tag{6}$$

Notice that $(x, q) \in S \times Act \times O \times Q$, hence an NGAC operation. While this approach is theoretically possible, in practicality, most implementations expect a total function. Then, we can add sink state that would be entered whenever a not allowed operation is encountered. If we denote this total function model with M_T, $M_T = (\Sigma, Q \cup \{q_{sink}\}, q_0, \delta_T, F$ where

$$\delta_T(q,x) = \begin{cases} q_{sink}, & q = q_{sink} \text{ else;} \\ \delta_U(q,\perp), & x = \perp \text{ else;} \\ \delta_U(q,x), & \text{policy allows } (x,q) \\ q_{sink}, & \text{otherwise} \end{cases} \tag{7}$$

It is important to note that M can be deterministic and non-deterministic depending on M_U.

VI. Case Study

To illustrate the benefits of using stateful policies, we will consider a fire response problem. Assuming a post-disaster situation, where many buildings in the city have caught fire, the response should be very organized. Firstly, high importance buildings like hospitals, warehouses, archives should be given priority. Also, more locally, if the fire in a part of the building is put out, the whole building should be secured to prevent fire from spreading to previously intervened part. For such things, it is useful to have an access control system that determines whether an action is allowed or not.

Using the framework that is described in section IV, we can model the problem as follows:

- $S = \{D1\}$
- $O = \{R_1, R_2, R_3, R_4, R_5\}$ where R_i denotes Room i. The enumeration of the rooms do not have a meaning.
- $Act = \{DropPayload\}$. In this simplified version, we will consider only a single action.
- $A_S = \{Drone\}$

We will explain the remaining part here, as they are too long to itemize.

Object attributes are as below. Note that B_i denotes Building i and $F_j(B_i)$ denotes Floor j of Building i.

$$A_O = \{CityA, HighImportance, LowImportance, B_1, B_2, B_3, F_1(B_1), F_2(B_1), F_1(B_2), F_1(B_3)\} \tag{8}$$

.

The attribute graphs G_S and G_O are shown in Figure 1.

There is only a general allowance rule that allows everything always: $Allow = \{(Drone, DropPayload, Q, CityA)\}$. But priority of this rule is 1.

There are two prohibition rules. The first one prevents the drone from going to low importance buildings before finishing all high importance ones: $(Drone, DropPayload, Q', LowImportance) \in Prohibit$. Note that $Q' \subseteq Q$ and Q' are the states where there is at least one $HighImportance$ room on fire.

The second one prevents the drone from leaving a building that it is in if there is a fire. These are actually a set of rules: $\{(Drone, DropPayload, Q_i, B_i) | 3 \geq i \geq 1\}$ where Q_i is the states where the drone in a building other than B_i and that building is not completely secured.

Each state consists of a room status, either $Fire$ or $NoFire$, for each room and the current position of the drone. More formally, $Q = \{Fire, NoFire\}^O \times O$. For this problem, we don't consider spread of the fire. Hence, state transitions model only the operated room becoming $NoFire$ after payload is dropped. If a more complex model is desired, a fire spread model can be developed. This would make the state machine non-deterministic but it is supported by the framework.

Note this problem definition focuses on the stateful nature of the problem. Hence, we have a single subject and action. The supplementary material also includes implementation of a Department access control which focuses on multiple subject and action aspect.

VII. Alloy Implementation

In this section, our approach to analyzing stateful policies in Alloy will be described. In order to maximize reusability of the code across different problem instances and classes, we did a modular implementation where each aspect of the problem is defined in a seperate file. This section will follow the same structure with the subsections Graph, Policy, State Machine, and Problem.

A. Graph: Describing the Graph

In this subsection, we will show how different components of attribute graphs can be implemented.

1) Introducing Subjects and Objects: We define a Subject as an abstract signatures with properties *NodeID* and *SubjectActions*. NodeID shows which

```
abstract sig Subject {
  NodeID: one AttributeNodeSubject,
  SubjectActions: set Actions
}
```

2) Describing the Attribute Graph: Graphs of subject and object attributes, G_S and G_O are defined separately and their nodes are represented as separate signatures.

The abstract *AttributeNodeSubject* as defined as:

```
abstract sig AttributeNodeSubject{
  children: set AttributeNodeSubject
}
```

Then, each of the vertices in G_S is a disjoint subset of this abstract signature. Notice that this signature makes the abstract signature just a union of the

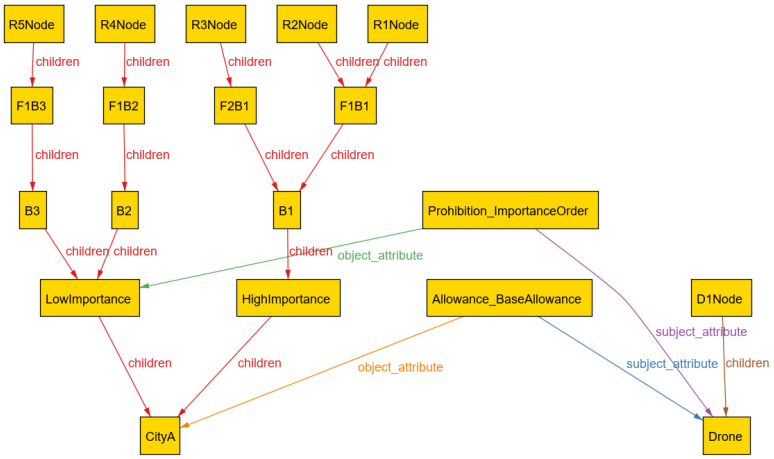

Figure 1: Attribute Graph for Fire Response as modelled in Alloy.

```
one sig D1Node extends AttributeNodeSubject {} {
  children = Drone
}
```

Now, we can write a predicate that checks whether an association path between a given subject and subject attribute exist in the graph.

```
pred SubjectDAGPath(sub: Subject,
                    sub_attribute: AttributeNodeSubject){
  (sub_attribute in sub.NodeID.*children)
}
```

B. Policy

In order to define a policy over the attribute graph, we introduce two more abstract signatures: *Allowance* and *Prohibition*. They are symmetric in their definitions; so, we will just focus on Prohibition signature.

```
abstract sig Prohibition {
  subject_attribute: AttributeNodeSubject,
  action_set:  set Actions,
  object_attribute: AttributeNodeObject,
  state_set: set State,
  priority: one Priority
}
```

Then, we can write concrete allowance rules by extending this abstract signature and binding concrete values to properties. We will show how this is done in the Problem section.

Now we will define policy access decisions.

```
1  pred access_prohibition(sub: Subject, actset: Actions,
2                          obj: Object, stateset: set State,
3                          p: Priority){
4    all act: actset, state: stateset |_
5    some snode: AttributeNodeSubject,
6      onode: AttributeNodeObject,
7        prohibit_rule: Prohibition |
8    SubjectDAGPath[sub, snode] and ObjectDAGPath[obj, onode] and
9    (snode=prohibit_rule.subject_attribute) and
10   (onode = prohibit_rule.object_attribute) and
11   (act in prohibit_rule.action_set) and
12   (state in prohibit_rule.state_set) and
13   (act in sub.SubjectActions) and
14   (act in obj.ObjectActions) and
15   (ord/lte[p, prohibit_rule.priority])
16 }
```

This predicate checks whether there is a prohibition rule for a given $(sub, actset, obj, stateset) \in$ tuple with a priority p or higher. We have a symmetric version for allowance rules too. Then, we can combine them to answer the access query:

```
1 pred access_check(sub: Subject, acts: Actions,
2                                 obj: Object, states: set State){
3   some p:  Priority|
4   access_allowance[sub, acts, obj, states, p] and
5   not access_prohibition[sub, acts, obj, states, p]
6   }
```

$access\_check$ checks whether the maximum priority of allowance rules is greater than the prohibition rules'.

C. State Machine

In this section, we will describe the environment model as a finite-state machine, so this part is problem specific.

```
1  module library/statemachine[subs, objs, acts]
2  open library/time
3  enum RoomStatus{Fire, NoFire}
4
5  // State Set
6  sig AutomatonState{
7    state_map :  objs -> one RoomStatus, // lookup table
8    curr: objs // current position of the drone
9    }
10 // Alphabet
11 sig Operation{
12   sub: subs,
13   act: acts,
14   obj: objs
15 }
16 // Initial State
```

```
fact InitialState{
(StateMachine.state.(time/first).state_map = objs -> Fire)
}
// Transition Function
fact TransitionFunction{
  (all t: Time | t = time/last or
  let s = StateMachine.state.t |
  let s' = StateMachine.state.(time/next[t]) |
  let i = StateMachine.input.t |
    (s'.state_map = s.state_map++(i.obj -> NoFire)) and
                  s'.curr = i.obj)
}
```

This state machine module is parameterized for graph description signatures; hence, it can take in any graph without changing import statements.

AutomatonState signature is the Q set of the state machine. It maps rooms to $Fire$ or $NoFire$ with $state\_map$ property and keeps track of the last room with $curr$. We define q_0 using $InitialState$ fact. Notice that $InitialState$ doesn't specify $curr$ property of the first state. It can be specified with a predicate separately or left unspecified to consider all cases.

Similarly, *Operation* signature is the alphabet,Σ, of the state machine.

$TransitionFunction$ fact describes the transition function δ by defining valid traces.

In order to answer some specific queries about this state machine, we add two other predicates: $startRoom$ and $roomExt$.

```
// Use this predicate to determine the room drone starts in.
pred startRoom(r: objs){
  StateMachine.state.(time/first).curr=r
}
// Check whether there is a time when
// there isn't a fire in the room.
pred roomExt(r: objs){
  some  t: Time |  StateMachine.state.t.state_map[r] = NoFire
}
```

D. Problem

Now that all elements of the policy are complete, we can put them together. We will firstly write the concrete rules for this problem.

```
one sig Allowance_BaseAllowance extends Allowance {} {
  (subject_attribute = Drone) and
  (object_attribute = CityA) and
  (action_set = Actions) and
  (state_set = AutomatonState) and
  (priority = P1)}
// Start With High Importance Buildings
one sig Prohibition_ImportanceOrder extends Prohibition {} {
  (subject_attribute = Drone) and
  (object_attribute = LowImportance) and
  (action_set = Actions) and
  (state_set = {aState: AutomatonState|
                        not AttributeEXT[HighImportance,
                                         aState] }) and
  (priority = P8)
}
// Check if all children of an attribute ext(inguished).
pred AttributeEXT(att: AttributeNodeObject,
                          aState: AutomatonState){
  all obj: Object |
      not ObjectDAGPath[obj, att] or
       aState.state_map[obj] = NoFire
}
// Don't leave until the building is done
one sig Prohibition_FinishBuildingB1_1 extends Prohibition{}{
  (subject_attribute = Drone)
  and (object_attribute = B2)
  and (action_set = Actions)
  and (state_set =  {aState: AutomatonState|
                       not AttributeEXT[B1, aState] and
                       ObjectDAGPath[aState.curr, B1]})
  and (priority = P7)
}
```

$Allowance\_BaseAllowance$ is a general allowance rule with a low priority $P1$ that means allow an action unless it is specifically prohibited. It uses the object attribute $CityA$, so it is valid for all objects.

$Prohibition\_ImportanceOrder$ imposes the rule of first saving the high importance buildings. It prohibits $LowImportance$ objects in states where there is a $HighImportance$ room that is on fire. This is checked by $AttributeEXT$ predicate, which checks if all objects with the given attribute have $NoFire$ status.

$Prohibition\_FinishBuildingB1\_1$ is actually just one of the rules for implementing "Don't leave until the building is finished" rule. We've left the rest out due to space constraints but they are similar. It is valid if the $curr$ is in $B1$ and $B1$ has a room that is on $Fire$. This would prohibit taking an action on a different building. This approach requires separate rules for each building. This works if there aren't many building in the problem like our case study.

An alternative approach for problems with many buildings is creating a $Building$ sub-signature of $AttributeNodeObject$. While this breaks the homogeneity of attribute nodes, it allows writing rules that is valid for all buildings and only for buildings.

After writing the policy rules, we should combine the policy with environment state machine. Imposing the policy on the state machine can be done by making sure that each input symbol is allowed by the policy. This is done by $validInputs$ predicate.

```
pred validInputs{
  all t:Time |
  let i = StateMachine.input.t |
  access_check[i.sub, i.act, i.obj,
                          StateMachine.state.t]
}
```

VIII. Conclusion

In this work, we have proposed an extended NGAC framework that both provides more flexibility when describing policies and remains tractable for automated analysis. Firstly, our framework supports multi-level hierarchies of allowance and prohibition rules. This allows having rules with different precedence levels. Also, our framework supports stateful policies which are dependent on an environment state. While this could be considered as just a collection of independent policies; we show that with a tractable environment model, such as a finite state machine, combining stateful policies with the environment model allows automated analysis of a dynamic system.

We presented our results on emergency fire response example where a policy is needed to organize an effective response that respects priorities and does not waste resources. We showed that we can create a non-deterministic finite state machine to model both the non-deterministic spread of the fire and impacts of user inputs like extinguishing fire in rooms. Then, we used this model with our policy to answer queries like whether the fire in a room could be put out in a given number of operation steps.

Because the main focus of this paper is offline audit of the system and configurations, time and scalability are not thoroughly investigated. This warrants further study of using Alloy for larger systems.

References

[1] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible access control markup language (xacml) and next generation access control (ngac)," in *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, 2016, pp. 13–24.

[2] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[3] W. Hassan, L. Logrippo, and M. Mankai, "Validating access control policies with alloy," in *Proceedings of the Workshop on Practice and Theory of Access Control Technologies, Quebec, Canada*, 2005, pp. 17–22.

[4] M. Toahchoodee and I. Ray, "Validation of policy integration using alloy," in *International Conference on Distributed Computing and Internet Technology*. Springer, 2005, pp. 420–431.

[5] P. Bennett, I. Ray, and R. France, "Modeling of online social network policies using an attribute-based access control framework," in *International Conference on Information Systems Security*. Springer, 2015, pp. 79–97.

[6] F. Nazerian, H. Motameni, and H. Nematzadeh, "Emergency role-based access control (e-rbac) and analysis of model specifications with alloy," *Journal of information security and applications*, vol. 45, pp. 131–142, 2019.

[7] K. K. Kolluru, C. Paniagua, J. van Deventer, J. Eliasson, J. Delsing, and R. J. DeLong, "An aaa solution for securing industrial iot devices using next generation access control," in *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, 2018, pp. 737–742.

[8] B. Bezawada, K. Haefner, and I. Ray, "Securing home iot environments with attribute-based access control," in *Proceedings of the Third ACM Workshop on Attribute-Based Access Control*, ser. ABAC'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 43–53. [Online]. Available: https://doi.org/10.1145/3180457.3180464

[9] R. Basnet, S. Mukherjee, V. M. Pagadala, and I. Ray, "An efficient implementation of next generation access control for the mobile health cloud," in *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, 2018, pp. 131–138.

[10] M. Sipser, *Introduction to the Theory of Computation*. Cengage learning, 2012.