Specification Of Access Control Features In Health Records

Project by,
Sai Venkatesh IMT2017012,
George Abraham IMT2017019,
Kaushal Mittal IMT2017024.
Under,
Prof. Sujit Kumar Chakrabarti

As a course project for,
CS 306 / Programming Languages

Table of Contents

1. Problem Statement

Specification of Access Control Features for Medical Records. (Using Athena)

1.1 Defining our problem statement:

In this project, we take a look at how to **formulate** and **analyze access-control policies** keeping in mind the policies for a hospital. We try to implement policies controlling the medical records of the patient ie: Allowing or denying requests made by doctors to access specific health records pertaining to a patient.

Access control is the process by which a system "controls which subjects (users) have access to which resources in the system". Policies are vital both to computer systems and to organizations such as companies, governmental institutions. A resource could be a body of information (database), a service, a physical object (a printer or a CPU), a certain quantity (bandwidth), and so on. Essentially, a resource is anything usable.

2. What is Athena?

Athena is a language for expressing proofs and computations. It is a **higher-order** language, meaning that procedures are first-class values that can be passed as (possibly anonymous) arguments or returned as results; it is **dynamically typed**, meaning that type checking is performed at run-time; it has side effects, it relies heavily on lists for structuring data, and those lists can be heterogeneous.

2.1 How to use Athena

domains: In Athena, a set of objects/class can be represented using the keyword "**domain**". For example, we can declare a domain called "Person", members of which are people. Domains can also be seen as **"sorts"**.

```
>(declare joe Person)

New symbol joe declared.

>(domain Person)

New domain Person introduced.
```

Function Symbols: After declaring domains, we can declare function symbols using the "**declare**" keyword, for instance:

```
>(declare father (-> (Person) Person))

New symbol father declared.
```

Here, we have declared a function symbol called “father”, which takes an input of type Person and produces another member of type Person. The **declare** keyword can also be used to declare constants such as members of Domain.

Procedures: A procedure is a lambda abstraction designed to compute complicated functions. For example

```
define (fact n) :=
  check {
    (less? n 1) => 1
  | else => (times n (fact (minus n 1)))
  }
```

Function symbols are not procedures. They are ordinary data values that can be manipulated by procedures.

Terms: A term is a syntactic object that represents an element of some sort. The simplest kind of term is a constant symbol. For instance, assuming the declarations of the previous section, if we type joe at the Athena prompt, Athena will recognize the input as a term

```
> joe

Term: joe
```

Variable: A variable is a term acting as a placeholder that can be replaced by other terms in appropriate contexts. They are denoted by “?” followed by a name.

```
> (father ?p)

Term: (father ?p:Person)
```

Propositions/Sentences:
Athena also provides the ability to write **propositions/sentences**, propositions are used to state relations between **“sorts”**. Athena allows to form three types of propositions:

- **Atomic propositions:** They are simply terms of sort Boolean
- Boolean combinations of propositions (allows the usage of combinators such as **not**, **and**, **or**, **if** and **iff**)
- **Quantified propositions** that make the use of quantifiers, The 2 types of quantifiers are **forall** and **exists**. It is the feature of quantifiers that makes Athena so powerful and write extremely expressive statements in first-order logic.

Example:

```
> (forall ?x:Person . ?x =/= father ?x)

Sentence: (forall ?x:Person
            (not (= ?x:Person
                    (father ?x:Person))))
```

Assumption Base: Athena maintains a global set of propositions called the **assumption base**. It is a set of propositions that we regard as true or a set of **axioms**. Every time an axiom is postulated or a **theorem** is proved at the top level, the corresponding sentence is inserted into the assumption base.

- (**assert (.....)**): To add a proposition/sentence/term to the assumption base, we use the command "**assert**".
- (**retract (.....)**) : This procedure can be used to remove specific sentences from the assumption base.
- (**show-assumption-base**): is a procedure that can be used to print all the sentences present in the assumption base.

```
> clear-assumption-base

Assumption base cleared.

> assert (1 = 2)

The sentence
(= 1 2)
has been added to the assumption base.

> retract (1 = 2)

The sentence
(= 1 2)
has been removed from the assumption base.
```

2.2 Theorem Proving Using Athena

At the beginning of section 2, we have defined ATHENA as a language for expressing proofs and computations. There are 2 methods for theorem proving **manual** and **automated methods**. Lets first see the manual method.

2.2.1 Manual Theorem Proving

The simplest type of proof is a single application of an **inference rule**. Inference rules in Athena are called **methods**. Athena comes with a small collection of predefined methods—the so-called **primitive methods**.

Inference Rules

Table 3.1.1 Rules of Inference

Rule of Inference	Tautology	Name
$\frac{p}{\therefore p \vee q}$	$p \rightarrow (p \vee q)$	Addition
$\frac{p \wedge q}{\therefore p}$	$(p \wedge q) \rightarrow p$	Simplification
$\frac{p \quad q}{\therefore p \wedge q}$	$((p) \wedge (q)) \rightarrow (p \wedge q)$	Conjunction
$\frac{p \quad p \rightarrow q}{\therefore q}$	$[p \wedge (p \rightarrow q)] \rightarrow q$	Modus ponens
$\frac{\neg q \quad p \rightarrow q}{\therefore \neg p}$	$[\neg q \wedge (p \rightarrow q)] \rightarrow \neg p$	Modus tolens
$\frac{p \rightarrow q \quad q \rightarrow r}{\therefore p \rightarrow r}$	$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$	Hypothesis syllogism
$\frac{p \vee q \quad \neg p}{\therefore q}$	$[(p \vee q) \wedge (\neg p)] \rightarrow q$	Disjunctive syllogism

- **(!Claim P):** This method takes an arbitrary proposition P as its sole argument and if P is in the assumption base, then claim simply returns P otherwise, if P is not in the current **assumption-base**, then claim reports an error.

  P = (male Joe), P is in the assumption base
  P1 = (and true (male Joe)), not in the assumption base
  Then (**! claim P1**) will result in an error.

- **left-and/right-and (Simplification):** If the conjunction is in the current assumption base, then the application **(!left-and (and P1 P2))** will produce P1 as the result; otherwise it will fail. Similarly **(!right-and (and P1 P2))** will produce P2 as the result.

- **both (conjunction introduction):** This is a binary method which takes two propositions P and Q and returns the conjunction (and P Q), provided that both P and Q are in the assumption base.

- **mp (conditional elimination):** mp is the abbreviation for modus ponens. It is a binary method **(!mp (if P Q) P)** which takes two propositions of the form (if P Q) and

P as arguments. If both of these are in the assumption base, the conclusion Q is returned.

- **either (addition):** This is a binary method **(!either P Q)** which takes two propositions P and Q and returns the disjunction (or P Q), provided that at least one of them is in the assumption base.

These are some of the built-in inferences which we can use in our theorem proving. Now we will see how to put together simple proofs/ inference rules to construct bigger proofs and we will also see some examples.

dseq ("deduction sequence"): In Athena, we can combine smaller proofs to make a bigger proof using **dseq.** For any two proofs D1 and D2, the phrase **(dseq D1 D2)** is a new composite proof that performs D1 and D2 sequentially. First, D1 is evaluated in the current assumption base, producing some conclusion P1. Then P1 is added to the assumption base, and we continue with the second proof D2. Thus D2 is evaluated in the newly updated assumption base. The result of D2 becomes the result of the entire **dseq**. Similarly for n > 2, **(dseq D1 · · · Dn).**

Now let's see 3 examples for better understanding:

- **A simple example:** Now we will prove that **(and A B)** is true then we will prove that **(and B A)** is also true.

eg.ath

```
1   (clear-assumption-base)
2
3   (declare (A B) Boolean)
4
5   (assert (and A B))
6
7   (dseq (!left-and (and A B))
8         (!right-and (and A B))
9         (!both B A)
10  )
```

```
Assumption base cleared.

New symbol A declared.

New symbol B declared.

The sentence
(and A B)
has been added to the assumption base.

Theorem: (and B A)
```

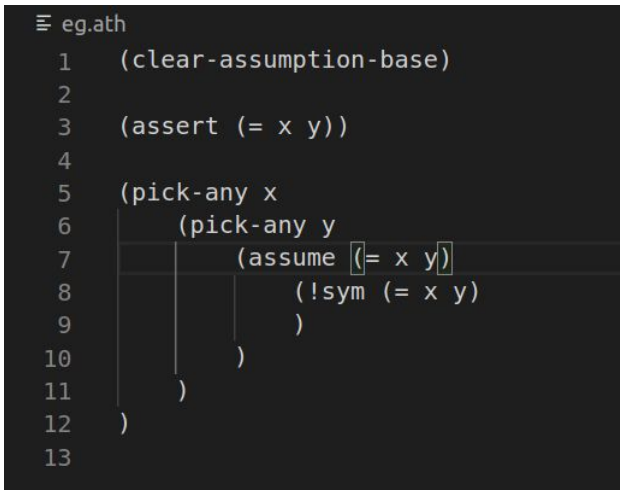
- **Conditional proof:** let's prove **(if (and A B) (and B A))**

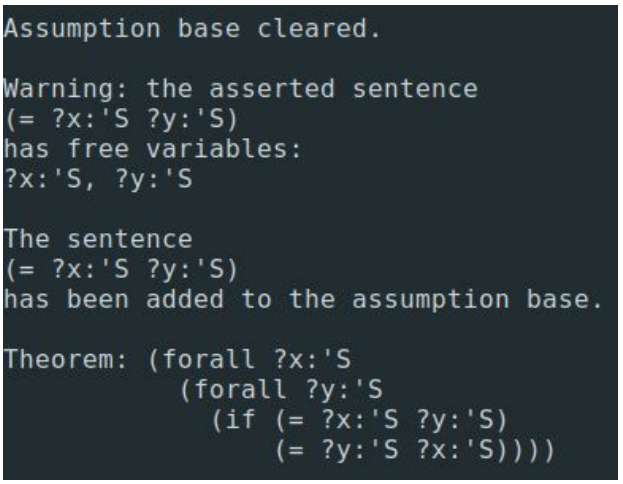
eg.ath

```
1   (clear-assumption-base)
2
3   (declare (A B) Boolean)
4
5   (assert (and A B))
6
7   (assume (and A B)
8       (dseq (!left-and (and A B))
9             (!right-and (and A B))
10            (!both B A)
11      )
12
13  )
14
```

```
Assumption base cleared.

New symbol A declared.

New symbol B declared.

The sentence
(and A B)
has been added to the assumption base.

Theorem: (if (and A B)
             (and B A))
```

- **Universal generalization:** now let's prove **(forall ?x ?y (if (= x y) (= y x)))**





2.2.2 Automated Theorem Proving

Constructing a realistic proof exclusively in terms of such inference rules can be very tedious. So Athena offers two key automation mechanisms

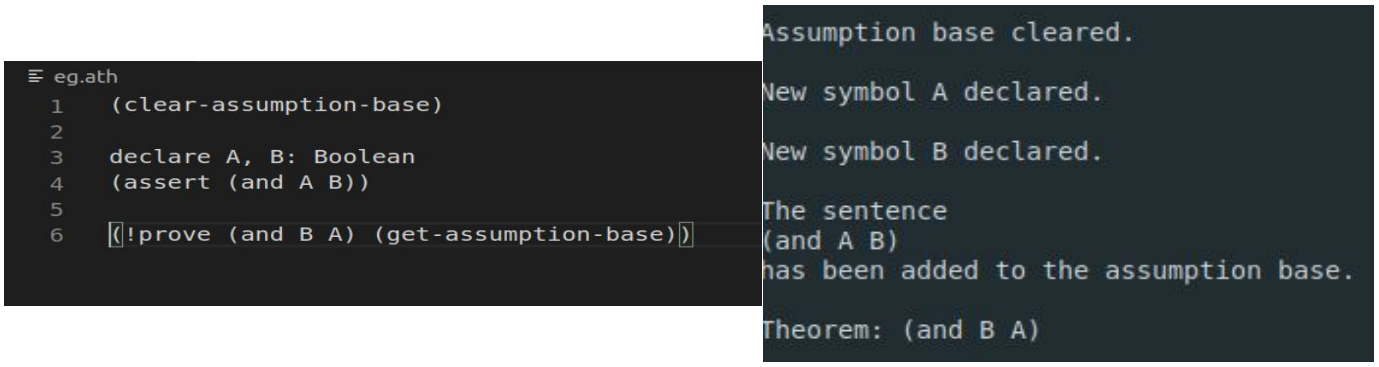
- **User-defined methods**
- **ATP:** Black-box facilities for automated theorem proving.

Now we move to the important feature of Athena which is the **automated theorem prover**. It is a black box-like facility which takes in a single proposition P and a list of propositions $[P_1 \cdots P_n]$ (the premises which have to be in the assumption base) and attempts to derive P from $[P_1 \cdots P_n]$. The theorem prover is called using the command "**!prove".** There are 3 possible outcomes on calling **prove**. So we can expect to see one of the following 3 results when we run our ATP based code.

- A **positive answer** is produced within the set time limit. It says that it was able to prove the proposition P using the given premises.
- A **negative answer** is produced within the set time limit. Tells us that given the current premises it was unable to conclude that P followed from $[P_1 \cdots P_n]$/goal does not, in fact, follow from the premises
- A **timeout occurs**. It might be the case that the theorem prover was not capable enough to come to conclusion P from $[P_1 \cdots P_n]$.

Let's see an example:
We will again prove **(if (and A B) (and B A)),** but unlike in proof mentioned in the conditional proof section, here we will use ATP

As we can see our 4 line proof became 1 line proof. Like this, we can use ATP to make



larger proofs more readable and less tedious and we can also use ATP as a stand-alone proof.

3. Athena for Access-Control

Specifications of access-control are needed to be defined as a set of rules called **"Policies"**. Policies determine whether a request raised by a subject must be allowed or denied.

3.1 Basic approach towards access control using automated theorem proving

We first take a look at the basic primitives involved in the Access control problem for a hospital.

- We have **doctors** and **patients**.
- Then we have **patient records** that are our resources.
- Every record is owned by a Patient.
- Every doctor belongs to a department.
- Every Patient is a **patient-of** particular Doctor. (implying this Patient's record is a part of this doctor's department).
- Some Doctors report to other Doctors. (defined as **Policy 1**) (if w1 ⇝ w2 then w2 can access records of patients who are patients of w1)
- **Records cannot be accessed by doctors from different departments unless allowed by the patient owning the record. (Policy 2)**
- Now we try to look at whether a medical professional **can access** a patient's medical record.

Policy 1
The reports to relation. Here we consider the professionals to belong to the **domain** of doctors. Suppose one healthcare professional w1 reports to another healthcare professional w2, then reports to relation is denoted by w1 ⇝ w2.

The rules that pertain to this policy is as follows :

- ⇝ relation is transitive. (w1 ⇝ w2) ∩ (w2 ⇝ w3) ⇒ (w1 ⇝ w3)
- ⇝ relation is antisymmetric. (w1 ⇝ w2) ⇒ ¬ (w2 ⇝ w1)
- (w1 ⇝ w2) ⇏ department (w1) = (department w2)
- (w1 ⇝ w2) ⇒ can_access (w1) ⊆ can_access (w2)

Clearly there is an inconsistency when both the policies come together
ie: If w1 ⇝ w2 where w2 belongs to a department different from w1, implies w2 can access records of patients-of w1 violating **Policy 2**.

3.1.1 Code Walkthrough

Instantiating primitives and policies

We declare the following domains: **Doctor, Patient**
Declare the following function symbols:

- **doctor_of : [Patient] ->Doctor** (patient as input and gives out doctor of that patient)
- **can_access: [Patient Doctor] -> Boolean** (returns Boolean value if doctor can access a patient's record)
- **reports: [Doctor Doctor] -> Boolean** (Boolean whether argument 1 reports to argument 2)

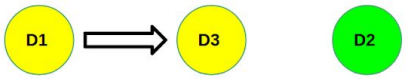
As we have the intention of finding inconsistencies between both the policies, first we assert the rules for Policy1 to the assumption base. Then we need to check if Policy 2 gets violated as an implication of Policy1 before adding it to the assumption base or else we would end up creating an erroneous composed policy. We model the same by defining the proposition **patient_violation** which tries to catch the violation and we try to prove the **patient_violation** given Policy1. If Athena is able to prove **patient_violation** using the premise of Policy1 that means that Policy2 is inconsistent with respect to Policy1 and should not be added to the assumption base.

Catching the inconsistencies:

First, we add the following function symbols and relations. We have 2 patients **patient1**, **patient2** and 3 doctors **doctor1**,**doctor2**,**doctor3**. **doctor1** is a doctor of **patient1** and so can access records of **patient1**. **doctor2** is a doctor of **patient2** and therefore, can access records of **patient2**. **doctor1** and **doctor3** are from **Department 1** whereas **doctor2** is from **Department 2**.

Let us consider 2 cases.

Case 1:



(Case 1: Consistent case)

When we add the relation that **doctor1** reports to **doctor3,** as we can see in the image, there is no instance of violation of Policy2 and therefore the automatic theorem prover of Athena is unable to prove **patient_violation**.

```
Unable to derive the conclusion
(exists ?y:Doctor
  (exists ?z:Doctor
    (exists ?h:Patient
      (or (and (in_Dep1 ?y:Doctor)
               (in_Dep2 ?z:Doctor)
               (reports ?y:Doctor ?z:Doctor)
               (= (doctor_of ?h:Patient)
                  ?y:Doctor)
               (can_access ?h:Patient ?z:Doctor))
          (and (in_Dep2 ?y:Doctor)
               (in_Dep1 ?z:Doctor)
               (reports ?y:Doctor ?z:Doctor)
               (= (doctor_of ?h:Patient)
                  ?y:Doctor)
               (can_access ?h:Patient ?z:Doctor))))))
```

*(Here Athena is unable to prove/find an instance of privacy violation, the statements seen above is the logic for the theorem of **patient_violation** for which Athena was unsuccessful in founding proof)*

Case 2:

(Case 2: Inconsistent case, because of cross department reporting)

When we add the relation that **doctor1** reports to **doctor3 (**and) **doctor3** reports to **doctor2,** as we can see in the image, there is a violation of Policy2 since a doctor of Department 2 (**doctor2**) is able to view the records of **patient1** without his permission and therefore the automatic theorem prover of Athena is able to prove **patient_violation** as shown in below image.

*(Athena proves an instance of privacy violation, the statements are seen above is the logic for the theorem of **patient_violation** for which Athena has successfully found a proof)*

```
Theorem: (exists ?y:Doctor
           (exists ?z:Doctor
             (exists ?h:Patient
               (or (and (in_Dep1 ?y:Doctor)
                        (in_Dep2 ?z:Doctor)
                        (reports ?y:Doctor ?z:Doctor)
                        (= (doctor_of ?h:Patient)
                           ?y:Doctor)
                        (can_access ?h:Patient ?z:Doctor))
                   (and (in_Dep2 ?y:Doctor)
                        (in_Dep1 ?z:Doctor)
                        (reports ?y:Doctor ?z:Doctor)
                        (= (doctor_of ?h:Patient)
                           ?y:Doctor)
                        (can_access ?h:Patient ?z:Doctor))))))
```

3.1.2 Advantages of the basic approach:

This model serves a few advantages and can be used in the following scenarios:

- Can be used for request evaluation by users
- Can be used to catch any instantiation of policy violation

3.1.3 Limitations of the basic approach:

We were able to catch an instance of policy violation and inconsistency through the above approach. The above implementation also allows for a good request evaluation given a set of policies. Still, there are severe limitations to this approach.

- It is unable to point to which property of policy leads to the **inconsistency** and reduces our ability for corrective measures.
- Checking whether there is a **redundancy** of a policy P (i.e if the addition of P to the set of policies S does not add any observable change.) is quite difficult
- Unable to find if a particular policy is **empty** (does not allow any requests) (or) whether the policy allows all requests (**completeness**)
- Suffers from the **state explosion problem.**

4. A More expressive approach

Sophisticated Access Control via SMT and Logical Frameworks
[KONSTANTINE ARKOUDAS, RITU CHADHA, and JASON CHIANG, ACM-2014]

4.1 Athena and SMT

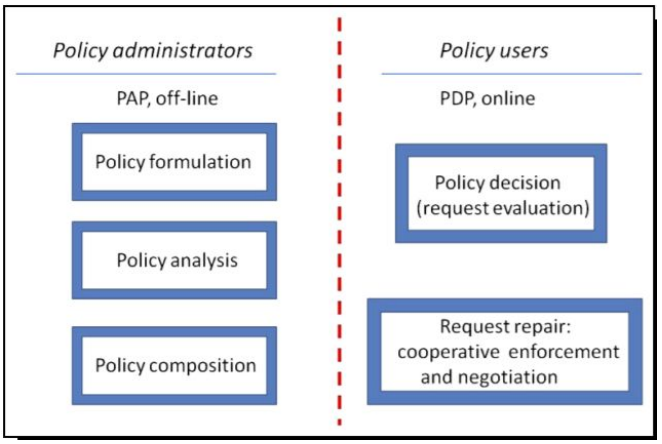
Below we define a few key points with respect to SMT and keywords that are heavily used as this paper reduce both request evaluation and policy analysis to SMT solving,

- **SMT(Satisfiability Modulo Theories)** solvers are similar to SAT solvers but can solve higher-level logics such as first-order logic. An SMT solver inputs **quantifier-free** first-order logic outputs either satisfiable (with a model that satisfies it) or unsatisfiable.
- The satisfiability of sentence p that is input to an SMT solver is then determined with respect to background theories, along with the Boolean structure of p
- **First-Order Logic** - is a generalization over Boolean logic including more complicated expressions such as constant, functions, and predicate symbols. First-order logic consists of logical symbols and nonlogical symbols (parameters). Eg - Logical Symbols - ¬, ∨, ∧, ∀, ∃ and Parameters - =, +, > , < , constants, etc.
- Athena provides an option to interface with two of the available SMT solvers namely, **Yices (v1) and CVC (cvc4)**. SMT functionality in Athena is built into the top-level module SMT.
- The SMT module is majorly divided into three parts. The first part smt-map converts the given input constraints to an intermediate language, the second part, based on the choice of solver, creates the input files and runs the particular solver's executable. The final part is collecting and giving the output or, writing the errors (if encountered any) to a file.
- Understanding how the SMT solver (Yices or CVC) works is currently out of scope.

4.2 Advantages of SMT over automated theorem proving

The previous approach (**Section 3**) had various shortcomings but they can be rectified with the approach mentioned in the paper (Sophisticated Access Control via SMT and Logical Frameworks [3]).

Access control system [3] discusses two important roles, **policy administrator** and **policy user** providing multiple functionalities to both under a single framework.



4.3 Closer look at policies (Criteria and demands to be satisfied)

4.3.1 Policy Analysis and Policy Interaction

Similar to the above approach, the general conceptual framework of access control proposed is that we have subjects (here we have doctors, receptionists, patients, etc), objects (the medical records), and actions (read a record / write to a record). To have a more robust scheme for Access control policy, we need to think of two sides of the policies, that is policy administrators and policy users. **Policy administrators** are responsible for designing, enforcing policies, and **analyzing policies**.

The SMT approach tries to satisfy the following **analysis** criteria:

- **Consistency** (no request is both permitted and denied by the policy.)
- **Coverage**. The emptiness and the completeness problem discussed in the previous section.
- **Conflict**. Does one policy conflict with another?
- **Change impact**. When a policy is modified, Is there an access request that was previously denied but is now either allowed or not covered? If so, find such a request.
- **The redundancy issue**
- **Verification.** Check whether a given policy satisfies some property. This is the most general analysis and will also be used for the above criteria.

Policy users interact with policies once they are set up and the requests of a user can be allowed/denied in accordance with the policies. Though the basic approach allows for request evaluation, it does not provide the necessary correction measures of why a medical professional was unable to access the file (and) what measures the user should take to gain access(in the previous case, had to take the permission of the patient) and respond with **optimal request repair**.

4.3.2 Interaction between policies

Another thing to note is that in many cases, more than one department in a hospital with different policies and base rules might have to come together(for example a gynaecology department may have a different set of governing rules compared to a general department). It can also extend to different organizations coming together, for example, a medical insurance company might need to interact with the hospital where both of them clearly have a different set of policies. In such a case, it requires an efficient manner of **policy composition** unlike the basic one where finding an inconsistency was a cumbersome task. SMT solvers allow for different mechanisms to integrate different policies and resolve their conflicts.

4.4 Core Concepts for Access Control Framework

There are primitive domains Subject, Object, and Action, Request. Every request has a subject, an object, and an action, which can be obtained by predefined function symbols as shown below

- **subject**: Request → Subject
- **object:** Request → Object
- **action:** Request → Action

There are also two predeclared function symbols allow and deny, with the following signature.

- **allow, deny:** Request → Boolean

These domains and function symbols constitute the fundamental conceptual framework of access policies. Specific actions can be introduced as constant symbols of type Action.

- **read, write:** Action

Subject roles can be represented via unary relations on Subject, such as what follows.

- **isDoctor, isSurgeon, isPatient:** Subject → Boolean

A particular access request can be expressed as a **sentence/proposition** specifying desired values for the subject, object, and/or action. For instance, consider a request for **doctor1** to write on patient record **record1**. We represent request as a constant symbol, let us call it **req**, of type Request, and we can specify the details of the request by a sentence such as the following.

- **subject(req) = doctor1 ∧ object(req) = record1 ∧ action(req) = write**

We write R(t) for a sentence specifying a request t, where t is a term of type Request.

4.5 Structure of Policies

Policies are either atomic (flat) or complex (nested). Complex policies contain other policies, which may themselves contain other policies, and so on, to an arbitrary nesting depth which enables to have extremely complex policies

4.5.1 Atomic Policies:

An atomic policy consists of:

- a **tag** to show it is an atomic policy
- a **name**
- a **rule base**: First, we need to understand what a rule is. It is a predicate that takes input a **term t** of type Request and produces a conditional sentence of the form
    - **p ⇒ allow(t)** [which is a permissive rule]
    - **p ⇒ deny(t)** [a prohibitive rule]

  where p is a **quantifier-free** SMT sentence. Example:
  λ r : Request.action(r) = read ⋀ ¬owns(subject(r),object(r)) ⇒ deny(r)
  (r is the Request. The rule denies a read request if the subject is not the owner of the object)
  A **rule base** consists of a **list of such access rules**.
- a **rule integrator**: It is a procedure that takes an arbitrary rule base and produces a policy basis using an integrating rule.
- a list of **environmental constraints**: It is a predicate similar to a rule, taking an arbitrary term t of type Request as input and constructing a sentence as output.
  An example would be:
  λ r: Request . isHeadsurgeon(subject(r)) ⇒ isDoctor(subject(r))
  (States hierarchy in the subject domain, if a person is a head surgeon in that department then he is also a doctor)
  Environmental constraints **need not include allow or deny**.
- a **policy basis**: It is defined as a list of two predicates [A D].The first predicate A characterizes all and only those requests that are allowed by the corresponding policy while the second predicate, D, characterizes the requests that are denied.

4.5.2 Complex Policies

A complex policy comes from the result of composing a number of other policies. The need for composing policies has already been discussed.

A complex policy P in our framework consists of the following components:

- **tag** identifying P as a complex policy

- **name**
- **list of component policies** [$P_1 \cdot \cdot \cdot P_n$] the children of the composed policy P where each P_i can be either atomic or complex (results in arbitrary nesting of policies).This is a **replacement for rule base in the atomic policy**.
- **basis combinator**: Instead of a rule integrator we have a basis combinator. Both perform very similar actions. A basis combinator is a procedure that takes a list of policy bases [[A_1 D_1] $\cdot \cdot \cdot$ [A_n D_n]] and produces a new policy basis [A D].
- a list of **environmental constraints**
- **policy basis**.

Let us take a closer look at how the basis combinator works. We define the addBases combinator which corresponds to the **permit-overrides algorithm** (whose logic will be explained shortly)

define addBases := λ [A_1 D_1] , [A_2 D_2] . makeBasis($A_1 \vee A_2$,($D_1 \vee D_2$) $\wedge$ ¬A_1 $\wedge$ ¬A_2)

Where **makeBasis** is a binary procedure that takes a permissive and a prohibitive predicate and simply puts them in a 2-element list.

define makeBasis := λ A, D . [A D]

This combinator produces a basis that permits a request if it is permitted by either input basis and denies a request if it is denied by either basis and permitted by neither. We can create similar basis combinators for any such set of governing rules similar to the **permit-overrides algorithm**.

The definition of '**addBases**' was initially applied to 2 policies and can also be done on a version that takes a list of bases as input as shown below:

λ bases . foldl(addBases, bases, emptyBasis)

foldl is the fold left version of the standard higher-order fold operator in functional programming.
(Note: An empty basis is an empty policy base. This is important when we apply a fold function on a list)

4.6 Request Evaluation

Request evaluation can be seen as follows: given a policy P and a request Q(t), return **'permit**, **'deny**, or **'na** on whether the policy allows, prohibits, or neither allows nor prohibits the request. Request evaluation can be seen as applying P to the given request.

- Policy P
- Request Q(t)
- Conjunction of all of P's environmental constraints C(t)

- Permissive predicates of P A(t)
- Prohibitive predicates of the basis of P D(t)

Possible Scenarios are ’permit, ‘deny or ‘na,

- If Q(t) ∧ C(t) ∧¬A(t) is unsatisfiable, then return ’permit
- Else if, Q(t) ∧ C(t) ∧¬D(t) is unsatisfiable, return ’deny
- Else, return ‘na

The reason that request analysis does not simply require consistency (if request r satisfied by A then accept or denied by D then deny) is that a single request can be consistent both with A and with D (referred to in the consistency problem below), and in such cases, the system should neither permit nor deny the request.

The authors achieve this by following the above procedure.

4.7 Policy Analysis

In this section, we take a look at a few policy analysis criteria.

(a) Property Verification:

A verification procedure ‘**verify’** takes a policy and a property and either verifies that the policy satisfies the given property or else produces a counterexample. The environmental constraints of the policy P are obtained through procedure **constraintsOf** that takes a policy P and returns a predicate representing the conjunction of all environmental constraints of P.

define verify := λ P, property . let property’ = constraintsOf(P) ∧ ¬property(P)
m = findModel(property’),
if m == none => (property hold), else m is counterexample

(b) Consistency: A policy is consistent iff, for any given request r, r is either denied or permitted (exclusively). Note that ‘**permits**’ talks about permitted predicate in policy P from A. Similarly ‘**denies**’ tells the denied predicates of request r from the policy basis of P from D.’**covers**’ tells if a request r is either permitted or denied by P. It basically tells if request r is covered by policy p. Defined as:

$$\lambda P . verify(P, \lambda P . \lambda r . \neg(permits(P, r) \wedge denies(P, r)))$$

(c) Coverage: Check if every request is either permitted or denied.Defined as:

λP . verify(P, λP . λ r . ¬covers(P, r))

(d) Observational Equivalence: Two policies P1, P2 are equivalent if the application of P_1

to r produces x then the application of P_2 to r also produces x.A simple trick to find if P_1 and P_2 are equivalent is to check a request r where two distinct outcomes x and y such that applying P_2 to r produces x but applying P_1 to r produces y. If such a case happens then P_1 and P_2 are not equivalent. Can be modeled by:

$$\lambda P_1,P_2.verify(P_1 , \lambda P . \lambda r . [permits(P,r) \Leftrightarrow permits(P_2,r)] \wedge [denies(P, r) \Leftrightarrow denies(P_2,r)])$$

In simple terms, the above expression checks that the permissive and prohibitive predicates of P_1, P_2 are respectively equivalent.

(e) Redundancy: In a list of policies $L = [P_1 \cdot \cdot \cdot P_{i-1} P_i P_{i+1} \cdot \cdot \cdot P_n]$ according to some policy-combining algorithm f (like **permit-overrides** that we have discussed in the complex policy section). To make sure that policy P_i is not redundant we can formulate the following statement

$$f([P1 \cdots Pn]) \sim f([P1 \cdots Pi{-}1\ Pi{+}1 \cdots Pn])$$

Here we check if the composition of policies [P1 ··· Pn] by applying f is equivalent to applying f on a list of policies [P1 ··· Pi−1 Pi+1 ··· Pn] (where we have removed ith policy). If both are equivalent, then P_i is redundant with respect to [P1 ··· Pi−1 Pi+1 ··· Pn].