

**The Open University**

**Dept. of Mathematics and Computer Science**

**Final Project**

**A Provenance-based Access Control Model**

**מנחה: פרופ' אהוד גודס**

**סמסטר ב' 2018**

**מגיש:**

**גיא בן מיור, ת.ז. 304850951**

**פרויקט מתקדם זה הוגש כחלק מהדרישות לקבלת תואר "מוסמך (M.Sc.) במדעי המחשב" באוניברסיטה הפתוחה**

**אוגוסט 2018**

# Table of Contents

[Table of Contents 2](#_Toc523933585)

[Background Information 3](#_Toc523933586)

[Project Description 4](#_Toc523933587)

[Project Purpose 4](#_Toc523933588)

[Project Scope and Requirements 4](#_Toc523933589)

[Provenance-based Access Control Model 4](#_Toc523933590)

[Provenance Data 4](#_Toc523933591)

[Model Overview 6](#_Toc523933592)

[Model Components 7](#_Toc523933593)

[Policy Specifications 9](#_Toc523933594)

[Model Assumptions and Weaknesses 10](#_Toc523933595)

[Implementation 10](#_Toc523933596)

[Architecture 10](#_Toc523933597)

[Complexity Cases 12](#_Toc523933598)

[User Interface 12](#_Toc523933599)

[Testing 13](#_Toc523933600)

[Case Studies Description 13](#_Toc523933601)

[Case’s JSON Format 16](#_Toc523933602)

[Project Technology 21](#_Toc523933603)

[Reliability 21](#_Toc523933604)

[Availability 21](#_Toc523933605)

[Maintenance 21](#_Toc523933606)

[Usable 21](#_Toc523933607)

[Software Dependents and IDE 21](#_Toc523933608)

[Running the Project 21](#_Toc523933609)

[Conclusions 22](#_Toc523933610)

[References 23](#_Toc523933611)

[Appendices 24](#_Toc523933612)

[Appendix A: COMPONENTS SPECIFICATION GRAMMAR 24](#_Toc523933613)

[Appendix B: POLICY SPECIFICATION GRAMMAR 25](#_Toc523933614)

[Appendix C: 𝐴𝑐𝑐𝑒𝑠𝑠𝐸𝑣𝑎𝑙𝑢𝑎𝑡𝑖𝑜𝑛(𝑎𝑢, 𝑎,𝑂) ALGORITHM 26](#_Toc523933615)

# Background Information

Provenance data of a system resource provides historical information about the resource’s pedigree, past activities occurred on the resource, etc. In the past, databases and other electronic information sources were trusted because they were under centralized control. However, nowadays data is often available on the internet without any centralized control of its integrity. The information sources (or, in case of a single large source, a part of it) may be inaccurate and contain low quality data, it is essential to provide provenance and other context information, which can help end users determine whether the query results are trustworthy [1]. This information is useful and has been demonstrated to be effectively usable in various computing systems, in different scientific as well as business application domains. Embedding provenance-awareness into systems has recently garnered a relatively large attention. Data warehouses and curated databases are some typical examples where provenance information can be essential. In both data warehouses and curated databases, a tremendous, and often manually, effort is used in the construction of the result database. First, by specifying the extract-transform-load (ETL) process and second, to add and update the database incrementally [2].

Provenance is the documentation of the origin of a data object and the processes that influence and result in a specific state of that object. Capturing and storing provenance data enables higher trustworthiness and usage rate of the underlying data [3]. One of the major concerns is how cyber security can be achieved and enhanced in a provenance-aware system. Security tasks include the utilization of information and knowledge of the provenance data, to address existing issues of insider threat detection, malicious data dissemination, etc. Where provenance data is more critical than the associated system data, it is also essential to secure the provenance data [2]. For example, the details on whether and how the content of an electronic document has changed since its creation, are important for keeping its integrity, which is essential for the security of the document [4].

Traditional access control mechanisms are built for specific purposes and complicated to configure, in order to address the complex demands associated with these new technologies. Provenance information can be used in different application domains, including access control. It can be used to implement new types of access control mechanisms, such as allowing users to access data if it was derived from a data they have created. Moreover, a provenance-based access control mechanism can implement new types of access control rules that can’t be implemented using traditional approaches [5]. Access control systems that are built upon provenance data will provide, by fully utilizing its unique characteristics, a foundation for new access control mechanisms that are highly capable of supporting features that are complicated to achieve by using traditional access control solutions.

Finding the best solutions for addressing the security issues in a provenance-aware environment, either by using provenance data to enhance the system security or by prioritizing the protection of the provenance data itself, remains a critical and exciting challenge for both academic as well as business communities.

# Project Description

## Project Purpose

The project’s purpose is to implement the base model of a Provenance-based Access Control Algorithm that focuses on a system captured and a system computable provenance data, along with object dependencies, and assumes a policy is given. The model described at “A Provenance-based Access Control Model” [3].

## Project Scope and Requirements

The project contains 2 parts - the server side and the client side:

Server side:

* Implements the algorithm – For every new request, returns the access evaluation.
* Stores all the provenance data – The data is composed of all the previous approved requests. In each new execution of the project, the data will be reset.
* Stores system’s policies – Two pre-defined use cases system (will be described later). In addition, the user able to update existing rules or upload a new policy rule.
* Provides information to the client side.

Client side:

* Provides a user-friendly interface to test the algorithm and allows the selection of a system and the creation of requests.
* Allows the auto execution of an initial request list for easy testing.
* Built as a web application for an easy multiplatform execution.

# Provenance-based Access Control Model

## Provenance Data

In any active system, a transaction uses subjects, objects, and the corresponding actions describing the interaction between them. The log of such transactions is visible as part of the basis of the provenance information. However, the benefits are limited without assigning relevant semantics to the transactions log. For transactions information to be considered useful provenance information, causality dependencies of transaction data should be utilized. Without causality dependency as semantics foundation, it is hard to utilize transaction flows and associated information.

The provenance data main components, as shown in figure 1, consist of artifacts, processes, and agents. The five main dependencies are defined as ‘used’ (process on artifact), ‘wasGeneratedBy’ (artifact on process), ‘wasControlledBy’ (process on agent), ‘wasDerivedFrom’ (artifact on artifact), and ‘wasTriggeredBy’ (process on process). Altogether, the components and dependencies form a directed acyclic graph, where the main components correspond to the nodes and the dependencies correspond to the edges.

We only utilize the direct dependencies of provenance data and omit the indirect dependencies of ‘wasDerivedFrom’ and ‘wasTriggeredBy’ in the implemented provenance data model. For example, an object may have the dependency ‘wasGeneratedBy(submit)’ with a submit process. That means that the object was generated by a submit action (as opposed to a grade action). In OPM, the dependency role is applicable to ‘used’, ‘wasGeneratedBy’ and ‘wasControlledBy’ dependencies.

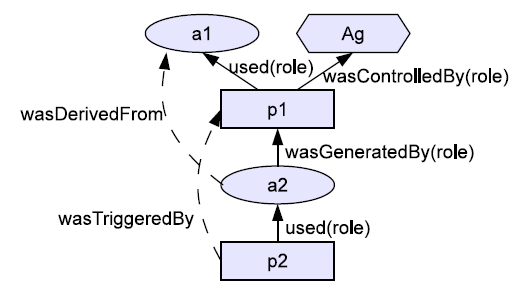


Figure 1. Provenance data model

Provenance data stores transactions data that are captured from performed actions. Each transaction is stored as a set of triples, consists of two entities and one causality dependency. The causality dependency utilizes three basic dependency types of ‘wasControlledBy’, ‘wasGeneratedBy’ and ‘used’.

For example, let’s assume a user 𝑢1 appended object 𝑜1 to object 𝑜2. The transaction data will be <𝑢1, 𝑎𝑝𝑝𝑒𝑛𝑑1, (𝑜1, 𝑜2), 𝑜1𝑣2 > and corresponding provenance data will store < 𝑎𝑝𝑝𝑒𝑛𝑑1, 𝑢1,’𝑤𝑎𝑠𝐶𝑜𝑛𝑡𝑟𝑜𝑙𝑙𝑒𝑑𝐵𝑦’ >, <𝑎𝑝𝑝𝑒𝑛𝑑1, 𝑜1, ‘𝑢𝑠𝑒𝑑’(𝑠𝑜𝑢𝑟𝑐𝑒) >, < 𝑎𝑝𝑝𝑒𝑛𝑑1, 𝑜2, ‘𝑢𝑠𝑒𝑑’(𝑟𝑒𝑓) > and < 𝑜1𝑣2, 𝑎𝑝𝑝𝑒𝑛𝑑1, ‘𝑤𝑎𝑠𝐺𝑒𝑛𝑒𝑟𝑎𝑡𝑒𝑑𝐵𝑦’(𝑎𝑝𝑝𝑒𝑛𝑑) >, where, 𝑜1 and 𝑜2 are input objects, 𝑜1𝑣2 is the output object and 𝑠𝑜𝑢𝑟𝑐𝑒, 𝑟𝑒𝑓, and 𝑎𝑝𝑝𝑒𝑛𝑑 are roles. The model does not allow roles for ‘wasControlledBy’ since we assume there is only one acting user per action instance. Figure 2 shows an example of the provenance data, along with the connection between the entities.

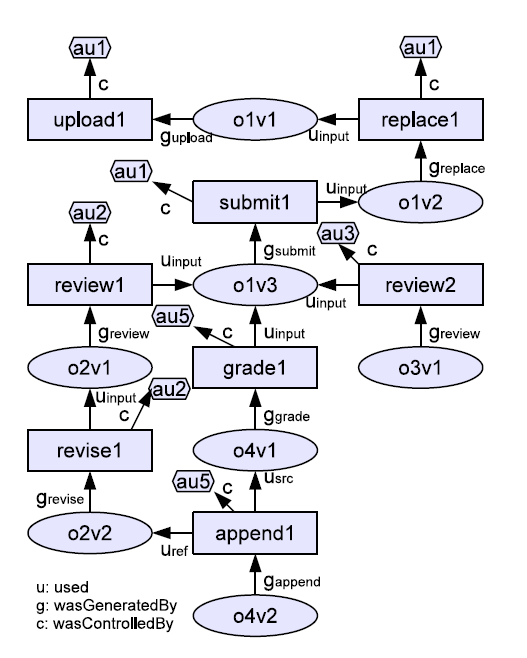


Figure 2. Sample Provenance Data in Graph

## Model Overview

The proposed provenance-based access control model consists of several core components. The purpose of the model is to decide on every transaction request, whether it’s allowed or not. Figure 3 shows these components, including acting users, action instances, action types, objects, object roles, provenance data, dependency lists, policies, access evaluation function for user authorization and action validation, policy retrieval policies, and policy retrieval function. The components with the solid line boundary are necessary for the proposed base model. (The ones with the dashed line boundary are part of the extended models)

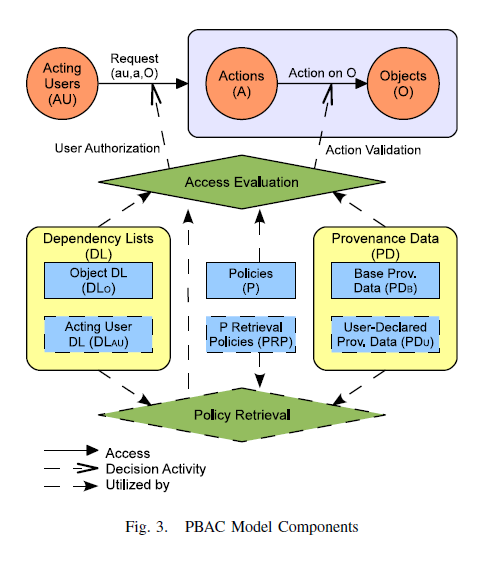


Figure 3. PBAC Model Components

## Model Components

* **Base Components:**
  1. **Acting Users (𝐴𝑈**) – Represent the human beings who initiate requests for actions against objects.
  2. **Objects (𝑂) -** Resource data that are accessed by users.
  3. **Object Roles (𝑂𝑅) –** Pre-defined specific object roles for each action type.
  4. **Action Instances (𝐴) -** Initiated by users when an object is being accessed, and of specific action type.
  5. **Action Types (𝐴𝑇) -** A fixed finite set of action types, predefined by system architects. We assume a system can derive action types from some given action instances. Each action type has a related policy that describes the cases in which it’s valid, and what object roles are related to the action.
  6. **Request - C**onsists of an acting user, an action instance and a set of objects with object roles that need to be accessed.
* **Provenance data (𝑃𝐷) Component:**

Consists of a base provenance data and a user-declared provenance data.

1. **Base Provenance Data (𝑃𝐷B)** – Further explained in section 3.1.
2. **User-Declared Dependency Data (𝑃𝐷U)** – If allowed, users can declare new dependencies or deny existing dependencies. It can be utilized in provenance-based access control, along with a base provenance data.

* **Dependency lists (𝐷𝐿) Component:**

List of pairs of abstracted dependency names (𝐷𝑁) and corresponding dependency path expressions (𝐷𝑃𝐴𝑇𝐻). Each 𝑑𝑛 ∈ 𝐷𝑁 is paired with, and defined by, exactly one 𝑑𝑝𝑎𝑡ℎ ∈ 𝐷𝑃𝐴𝑇𝐻, where 𝑑𝑝𝑎𝑡ℎ may use other previously defined 𝑑𝑛’s.

1. **Dependency Path (𝐷𝑃𝐴𝑇𝐻)** – Built from a sequence of path rule, other dependency path and connecting operators. It can be used in a reverse order. The dependency path describes the path that should be tracked on in the provenance data graph.
2. **Path Rule** – Contains an object rule and a provenance dependency type.
3. **Dependency Operators –C**onnects between the dependency paths and/or the path rules on the same dependency path. It can be: OR "|", GROUP\_START "(", GROUP\_END ")", ZERO\_OR\_MORE "\*", ZERO\_OR\_ONE "?", ONE\_OR\_MORE "+", ANY\_OBJECT ".".

Examples of dependency lists:

* + - 1. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑠𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑟𝑒𝑣𝑖𝑠𝑒. 𝑢𝑖𝑛𝑝𝑢𝑡 >- The path starts with a generated revised and then to used input. Described by the green line on figure 4 that starts with o2v2.
      2. < 𝑤𝑎𝑠𝑂𝑛𝑒𝑂𝑓𝑅𝑒𝑣𝑖𝑒𝑤𝑂𝑓, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑠𝑒𝑑𝑉 𝑜𝑓 ∗. 𝑔𝑟𝑒𝑣𝑖𝑒𝑤. 𝑢𝑖𝑛𝑝𝑢𝑡 > - The path starts with a zero or more times of (a) path, goes to generated review and then to used input. Described by the red line on figure 4 that starts on o2v2, and blue line on figure 4 that starts with 03v1.

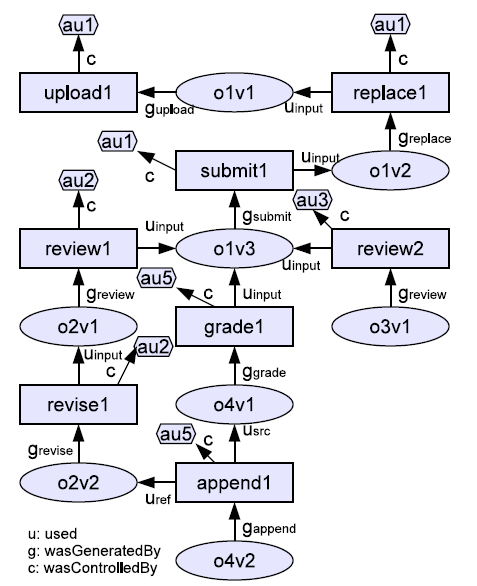


Figure 4. Sample Provenance Data for Online Grading System

* **Policies Component:**

Include a set of rules that need to be evaluated for granting access purposes. It’s built from a user authorization rules set and an action validation rules set. Each set of rules can be paired with ∧ and ∨ operators. Furthermore, each set can be built by combining other sets or any combination of a single rule with the same type.

1. **User Authorization Rule –**Specifies whether the requester is authorized to make the request. It describes whether the requester user should be included on a specific dependency path.
2. **Action Validation Rule –** Specifies whether the requested action can be performed against the requested objects. The comparison can be performed in one of two forms:
   1. **Number Comparable** – Defines whether the number of objects that comes from the dependency path is equal/not equal/big/big equal/small/small equal to a specific number.
   2. **Another Object Comparable** – Defines whether the list of objects that comes from the dependency path is equal/not equal/contained to the list of objects that comes from a second dependency path.

Examples of policies:

* + 1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑟𝑒𝑝𝑙𝑎𝑐𝑒, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉𝑜𝑓) ∣ = 0.

au user can replace object o if o was authored by au and o hasn’t submitted yet.

* + 1. 𝑎𝑙𝑙𝑜𝑤(𝑎𝑢, 𝑟𝑒𝑣𝑖𝑠𝑒, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜,𝑤𝑎𝑠𝐶𝑟𝑒𝑎𝑡𝑒𝑑𝑅𝑒𝑣𝑖𝑒𝑤𝐵𝑦) ∧

∣ (𝑜, 𝑤𝑎𝑠𝑂𝑛𝑒𝑂𝑓𝑅𝑒𝑣𝑖𝑒𝑤𝑂𝑓. 𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓−1) ∣ = 0.

Au user can revise object o if o was created by au and o is either a revise of a review or a review of a non-graded homework.

* **Access Evaluation:**

Evaluates a request by utilizing user authorization rules and action validation rules, which was found in the policy for the action type of the requested action and returns a Boolean value.

Appendix A contains a more detailed components specifications review.

## Policy Specifications

Policies for the proposed model consist of a set of user authorization rules (𝑈𝐴𝑅𝑢𝑙𝑒𝑠) and action validation rules (𝐴𝑉 𝑅𝑢𝑙𝑒𝑠). The overall result is the conjunction of the sub-results. Each rule is defined using path rules that consist of a starting node and a dependency name to which a regular expression-based dependency path pattern is mapped in a dependency list.

A user authorization rule is defined by an acting user, a path rule and an operator, and checks the existence of the acting user in the vertices found using the path rule. The action validation rule is defined by one or two of the path rules and an operator, and either checks the existence or frequency of the vertices in the path, or compares two sets of vertices found, one for each path. These three types of rules (one user authorization rule and two action validation rules) are by no means exhaustive but are sufficient to capture the sample use case scenario presented in this paper. Each user authorization rule is individually evaluated to a Boolean result. The individual results are then combined using disjunction and conjunction as specified. Action validation rules are similarly individually evaluated and then the results are combined using disjunction and conjunction as specified.

Appendix B contains a more detailed specifications review.

## Model Assumptions and Weaknesses

The implemented provenance data basic model adopts roles for ‘used’ and ‘wasGeneratedBy’ dependencies only, as it allowed only one agent for each process. In addition, as mentioned earlier, the model does not allow roles of ‘wasControlledBy’ since we assume there is only one acting user per action instance. Furthermore, the model considers object dependency only and does not involve user-declared dependency data.

Although provenance-based access control utilizes provenance data to make access decision, it is likely that a real-world system will also require other forms of access control systems along with PBAC. For example, consider a homework review and grading example in an online course management system presented in our sample case study. PBAC

supports only policies in which “the user who uploaded the homework is the only one who can replace it with a newer version or submit it”,” the user who submitted the homework can’t review it”, or “a user can append reviews to a grade report only if the homework review was completed”. This application system is likely to utilize more role-based access control, to enforce policies such as “only students can submit homework or review other student’s homework” and “only instructors can grade a homework or append reviews to a grade report”. For that reason, the policies in the proposed model are the necessary rather than the sufficient policies implemented for access control, since additional non-provenance-based policies may be also implemented.

# Implementation

## Architecture

The project contains 2 parts, the server side and the client side. The PBAC model implemented on the server side and the client-side is mainly used for testing.

The server side is divided into 6 different packages:

1. base components class – all base class which described on model base components that are needed for the model.
2. dependency component class – all class that are related to the dependency list.
3. provenance component class – all class that are used to configure a provenance data record.
4. policy component class – all class for supporting policy definition.
5. System cases loader – use cases systems to be tested.
6. Server application – built on Spring web framework (service and controller), to provide information to the client side.

The cases data (dependencies and policies) is loaded into the memory during the server startup and can be replaced and updated from the client. Every time a case is loaded, the related provenance data is initiated, and stored in the memory.

* 1. Important Classes and Methods Description

1. class Provenance
   * 1. Contains the ‘wasControlledBy’, ‘wasGeneratedBy’ and ‘UsedBy’ dependencies.
     2. Each dependency type has its own class with the relevant properties.
     3. The constructor receives an approved transaction instance and builds the dependencies.
     4. Can restore the relevant dependencies by action, object and user in a regular and reverse direction.
2. class DependencyPath
3. Contains a list that represents the sequence that builds the path. The sequence should be built only from PathRule, another DependencyPath and DependencyOperator. Refer to rule 6 in section 3.4, in Model Specifications.
4. Has a direction. Every path can be requested in a direct or a reverse order.
5. public List<Base> getObjectsByPath (DependencyList dependencyList,

List<Provenance> provenanceData,

List<? extends Base> inputObjects)

A method that receives the system dependency list, previous provenance data and objects.

It runs over the path (in a direct or a reverse order) on the provenance data graph for each input object.

The method should relate to all the different possibilities of DependencyOperator and sub-complexes DependencyPath.

1. class PathRule
2. Contains an object role and a dependency type.
3. public List<Base> find (List<Provenance> provenanceData,

Base inputObject, boolean isReverse)

This method checks whether the input object is mentioned on the relevant provenance data.

1. class Policy
2. Contains a UserAuthorizationRules and ActionValidationRules, representing a set of rules, which the policy is ruled by.
3. Each rule has a validation method that checks the dependency based on the dependency path result.
4. class JsonCaseStudy extends CaseStudy
5. All case studies systems, loaded from a json file.
6. class AccessEvaluation
7. Evaluates a request by utilizing user authorization rules and action validation rules found in the policy for the action type of the requested action and returns a Boolean value.
8. This class implements the main algorithm in the model as shows in Appendix C.

## Complexity cases

The project implementation supports all complexity model specifications, such as:

1. 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 ::= < 𝑈𝐴𝑅𝑢𝑙𝑒 > ∣ “(” < 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 > “)” ∣ < 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 >< 𝐶𝑜𝑛𝑛𝑒𝑐𝑡 >< 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 >

Each 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 can be built from an unlimited number of 𝑈𝐴𝑅𝑢𝑙𝑒𝑠 with connecting sub 𝑈𝐴𝑅𝑢𝑙𝑒𝑠.

1. 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 ::= < 𝐴𝑉𝑅𝑢𝑙𝑒 > ∣ “(” < 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 > “)” ∣ < 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 >< 𝐶𝑜𝑛𝑛𝑒𝑐𝑡 >< 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 >

Each 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 can be built from an unlimited number of 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠 with connecting sub 𝐴𝑉 𝑅𝑢𝑙𝑒𝑠.

1. 𝐴𝑉𝑅𝑢𝑙𝑒: = “∣” < 𝑃𝑎𝑡ℎ𝑅𝑢𝑙𝑒 > “∣” < 𝑜𝑝𝑒𝑟2 >< 𝑁𝑢𝑚𝑏𝑒𝑟 > ∣ < 𝑃𝑎𝑡ℎ𝑅𝑢𝑙𝑒 >< 𝑜𝑝𝑒𝑟3 >< 𝑃𝑎𝑡ℎ𝑅𝑢𝑙𝑒 >

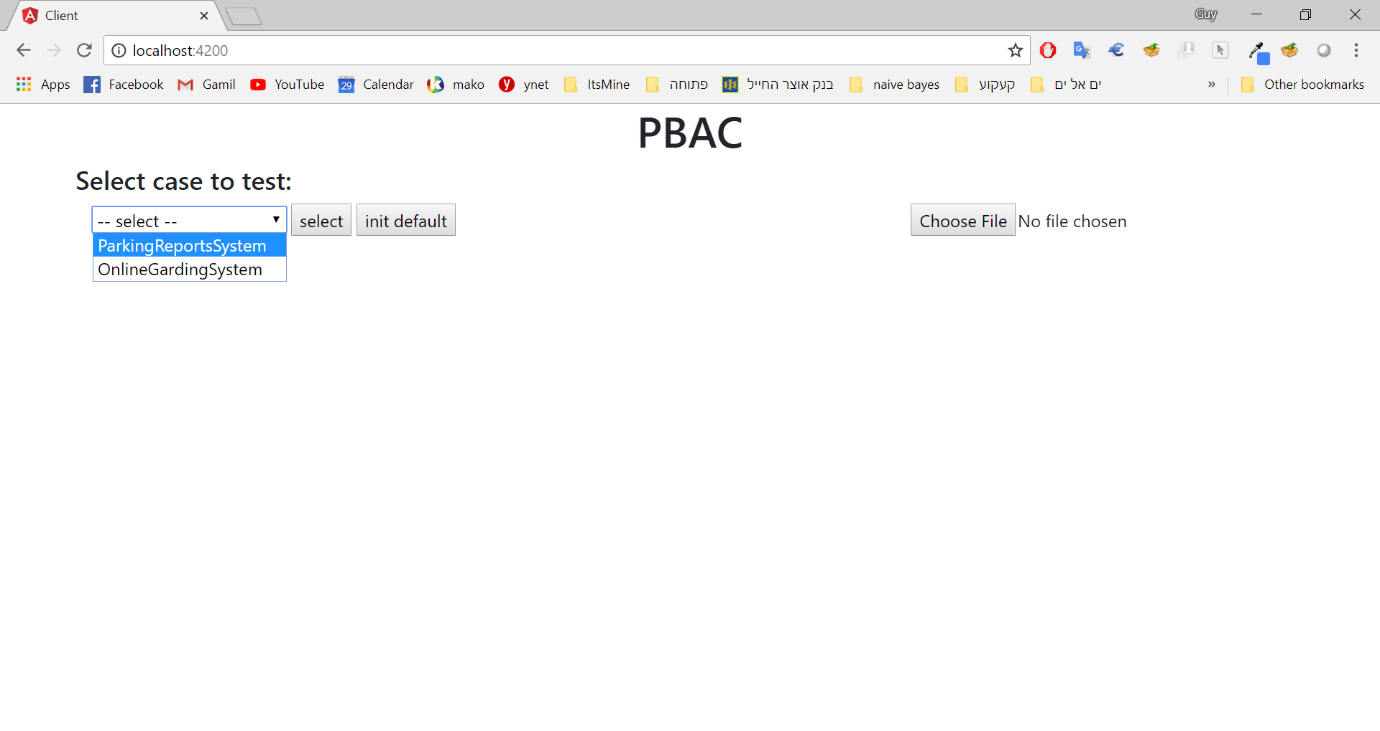
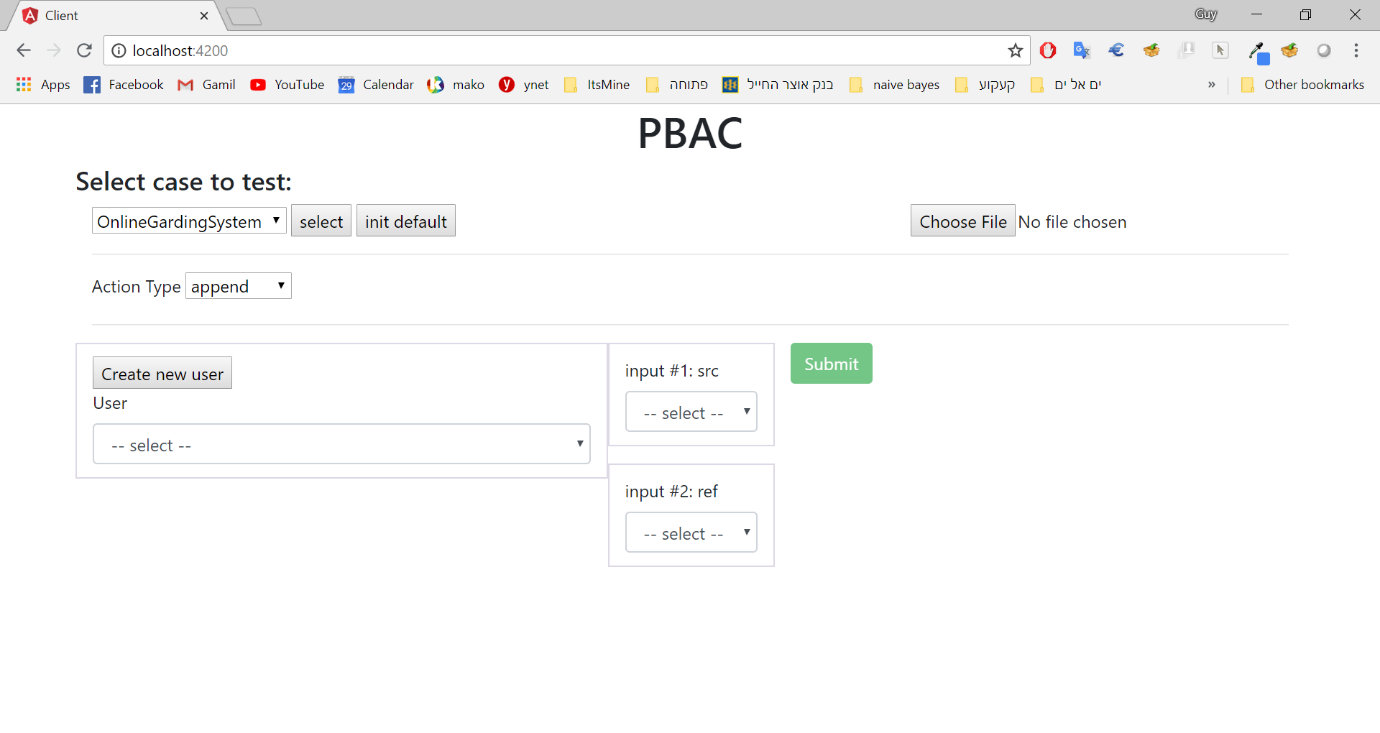
An 𝐴𝑉𝑅𝑢𝑙𝑒 is one of two types of roles, a number base or another path𝑅𝑢𝑙𝑒 comparable.

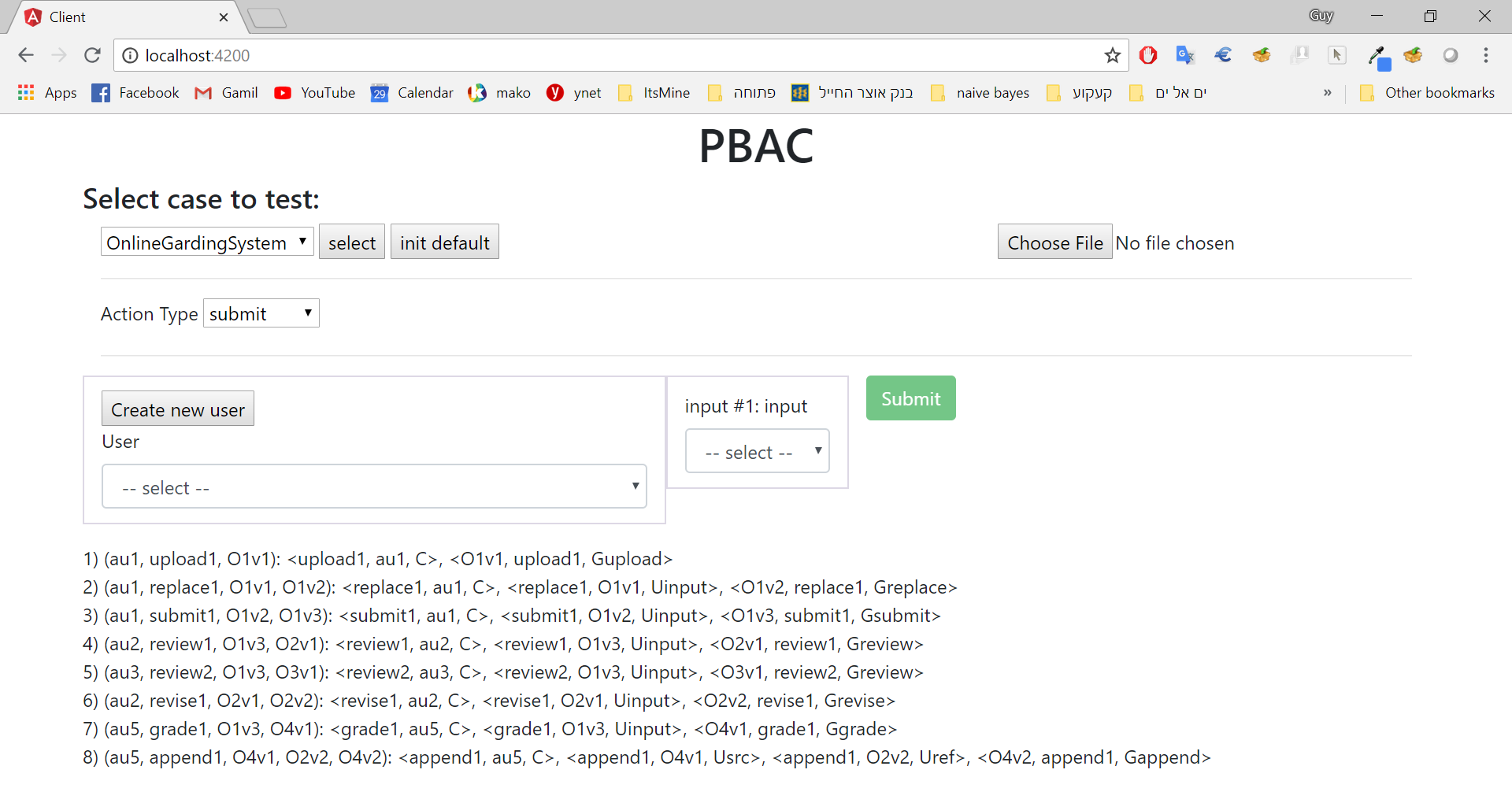
1. ∀𝑝 ∈ Σ, 𝑝 ∈ 𝐷𝑃𝐴𝑇𝐻; 𝜖 ∈ 𝐷𝑃𝐴𝑇𝐻; (𝑃1∣𝑃2), (𝑃1. 𝑃2), 𝑃1∗, 𝑃1+, 𝑃1? ∈ 𝐷𝑃𝐴𝑇𝐻, where 𝑃1 ∈ 𝐷𝑃𝐴𝑇𝐻 and 𝑃2 ∈ 𝐷𝑃𝐴𝑇𝐻.

Each dependency path can be built from an unlimited number of dependency paths with connecting sub dependency path, with 5 different implementations every time.

## User interface

The user interface is a single-page web application that has multiple views:

1. Case studies system select:
   1. The user can upload a new case study or update existing case study.
   2. The user can select a system to test the algorithm on.
   3. The user can auto-execute a list of requests to build the previous provenance data records.
2. After the user has selected a case, the system will load the available action types for the server, so the user will be able to create the requests.
   1. The user can choose any of the actions types that has been configured in the case study.
   2. The user can create a new acting user or select an existing one to perform the request.
   3. For every action type that has been selected, the system will show a different number of select box for input objects. The number of input objects is pre-configured in the use case json configuration.
3. The system will show a list of all transactions and related provenance data that has been created until the current time, based on the previous requests.



## Testing

To verify the project accuracy, the project will pre-bundle with 2 case study examples. Each case study describes the system objects policies. Each case study contains a dependency list and a list of action types with their corresponding policy. The case study is provided in a json format for an easy modifying and testing. The project will include an optional initial provenance data of easy testing.

### Case studies description

1. **Homework grading system**
   1. **Policies:**
2. Anyone can upload a homework document.
3. A user can replace an old version of a homework document with a newer version (versioning control), only if the user is the author of the old version and the old version has not been submitted yet.
4. An author can submit the homework (origin-based control), only if it was not submitted already.
5. A user can review a submitted homework only (work-flow control), only if the user is neither the author nor one of the existing reviewers of the homework (dynamic separation of duty), and the homework has been reviewed less than 3 times and hasn’t been graded yet.
6. A review can be revised by a user, only if the user created the review and the referred homework hasn’t been graded yet.
7. A homework document can be graded only if it was reviewed at least 2 times.
8. A review can be appended into a grade only if the acting user created the grade and the review was made for the homework that was graded.

* 1. **Dependency list:**

1. < 𝑤𝑎𝑠𝑅𝑒𝑝𝑙𝑎𝑐𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑟𝑒𝑝𝑙𝑎𝑐𝑒.𝑢𝑖𝑛𝑝𝑢𝑡 >
2. < 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑠𝑢𝑏𝑚𝑖𝑡.𝑢𝑖𝑛𝑝𝑢𝑡 >
3. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓, 𝑔𝑟𝑒𝑣𝑖𝑒𝑤.𝑢𝑖𝑛𝑝𝑢𝑡 >
4. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑠𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑟𝑒𝑣𝑖𝑠𝑒.𝑢𝑖𝑛𝑝𝑢𝑡 >
5. < 𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓, 𝑔𝑔𝑟𝑎𝑑𝑒.𝑢𝑖𝑛𝑝𝑢𝑡 >
6. < 𝑤𝑎𝑠𝐴𝑝𝑝𝑒𝑛𝑑𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑎𝑝𝑝𝑒𝑛𝑑.𝑢𝑠𝑟𝑐 >
7. < 𝑤𝑎𝑠𝑂𝑛𝑒𝑂𝑓𝑅𝑒𝑣𝑖𝑒𝑤𝑂𝑓, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑠𝑒𝑑𝑉 𝑜𝑓 ∗ .𝑔𝑟𝑒𝑣𝑖𝑒𝑤.𝑢𝑖𝑛𝑝𝑢𝑡 >
8. < 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓?. 𝑤𝑎𝑠𝑅𝑒𝑝𝑙𝑎𝑐𝑒𝑑𝑉 𝑜𝑓 ∗.𝑔𝑢𝑝𝑙𝑜𝑎𝑑.𝑐 >
9. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝐵𝑦, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓−1. 𝑔𝑟𝑒𝑣𝑖𝑒𝑤.𝑐 >
   1. **Policies in formal presentation:**
10. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑢𝑝𝑙𝑜𝑎𝑑, 𝑜) ⇒ true.
11. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑟𝑒𝑝𝑙𝑎𝑐𝑒, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑠𝑢𝑏𝑚𝑖𝑡, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑟𝑒𝑣𝑖𝑒𝑤, 𝑜) ⇒ 𝑎𝑢 ∉ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

𝑎𝑢 ∉ (𝑜,𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓) ∣ ≠ 0 ∧ ∣ (𝑜, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓−1) ∣ < 3 ∧

∣ (𝑜, 𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓−1) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤(𝑎𝑢, 𝑟𝑒𝑣𝑖𝑠𝑒, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜,𝑤𝑎𝑠𝐶𝑟𝑒𝑎𝑡𝑒𝑑𝑅𝑒𝑣𝑖𝑒𝑤𝐵𝑦)

∣ (𝑜, 𝑤𝑎𝑠𝑂𝑛𝑒𝑂𝑓𝑅𝑒𝑣𝑖𝑒𝑤𝑂𝑓. 𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓−1) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑔𝑟𝑎𝑑𝑒, 𝑜) ⇒ ((∣ (𝑜, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓−1) ∣ ≥ 2 ∧

∣ (𝑜, 𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓−1) ∣ = 0).

1. 𝑎𝑙𝑙𝑜𝑤(𝑎𝑢, 𝑎𝑝𝑝𝑒𝑛𝑑, 𝑜𝑠𝑟𝑐, 𝑜𝑟𝑒𝑓 ) ⇒ 𝑎𝑢 ∈ (𝑜𝑠𝑟𝑐,𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝐵𝑦) ∧

(𝑜𝑠𝑟𝑐,𝑤𝑎𝑠𝐺𝑟𝑎𝑑𝑒𝑑𝑂𝑜𝑓) = (𝑜𝑟𝑒𝑓 ,𝑤𝑎𝑠𝑂𝑛𝑒𝑂𝑓𝑅𝑒𝑣𝑖𝑒𝑤𝑂𝑓).

For example, transaction can be described as: (𝑎𝑢2, 𝑟𝑒𝑣𝑖𝑒𝑤1, 𝑜1𝑣3) – user 2 can review the homework object 1 and version 3.

1. **Parking reports system**
   1. **Policies:**
2. Inspector can upload a new parking report
3. A user can appeal on parking report if the user is not the author of the report.
4. An appeal can be created on report, and only once.
5. The user can append documents and images to the appeal, only if the user created it and the appeal hasn’t been submitted yet.
6. An appeal can be submitted only once and only by its author.
7. A user can review only a submitted appeal that was authored by others, and not the user isn’t the author of the related report.
8. A user can review an appeal only once.
9. Every appeal can be reviewed up to 2 times, as long as no review result has been set.
10. Only the user that created a review can determine if the appeal has been approved.
    1. **Dependency list:**
11. < 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦, 𝑔𝑢𝑝𝑙𝑜𝑎𝑑. 𝑐 >
12. < 𝑤𝑎𝑠Appeal𝑂𝑜𝑓, 𝑔appeal. 𝑢𝑖𝑛𝑝𝑢𝑡 >
13. < 𝑤𝑎𝑠𝐴𝑝𝑝𝑒𝑛𝑑𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑎𝑝𝑝𝑒𝑛𝑑. 𝑢𝑖𝑛𝑝𝑢𝑡 >
14. < 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓, 𝑔𝑠𝑢𝑏𝑚𝑖𝑡.𝑢𝑖𝑛𝑝𝑢𝑡 >
15. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓, 𝑔𝑟𝑒𝑣𝑖𝑒𝑤.𝑢𝑖𝑛𝑝𝑢𝑡 >
16. < 𝑤𝑎𝑠Determined𝑉 𝑜𝑓, 𝑔determine. 𝑢𝑖𝑛𝑝𝑢𝑡 >
17. < 𝑤𝑎𝑠Appealed𝐵𝑦, 𝑤𝑎𝑠Submitted𝑉 𝑜𝑓?. 𝑤𝑎𝑠Appended𝑉 𝑜𝑓∗. 𝑔𝑎𝑝𝑝𝑒𝑛𝑑.𝑐 >
18. < 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝐵𝑦, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝑂𝑜𝑓−1. 𝑔𝑟𝑒𝑣𝑖𝑒𝑤.𝑐 >
19. < 𝑤𝑎𝑠Uploaded𝑂𝑜𝑓, 𝑔𝑢𝑝𝑙𝑜𝑎𝑑 >
    1. **Policies in formal presentation:**
20. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑢𝑝𝑙𝑜𝑎𝑑, 𝑜) ⇒ true.
21. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, appeal, 𝑜) ⇒ 𝑎𝑢 ∉ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, wasAppealedOof−1) ∣ = 0 ∧ ∣ (𝑜, 𝑤𝑎𝑠Uploaded𝑂𝑜𝑓) ∣ = 1.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, append, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠Appealed𝐵𝑦) ∧

∣ (𝑜, wasAppealedOof−1) ∣ = 0 ∧ ∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, 𝑠𝑢𝑏𝑚𝑖𝑡, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, wasAppealedOof−1) ∣ = 0 ∧ ∣ (𝑜, 𝑤𝑎𝑠𝑆𝑢𝑏𝑚𝑖𝑡𝑡𝑒𝑑𝑉 𝑜𝑓) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, review, 𝑜) ⇒ 𝑎𝑢 ∉ (𝑜, 𝑤𝑎𝑠𝐴𝑢𝑡ℎ𝑜𝑟𝑒𝑑𝐵𝑦) 𝑎𝑢 ∉ (o, 𝑤𝑎𝑠Appealed𝐵𝑦) ∧

𝑎𝑢 ∉ (o, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝐵𝑦) ∣ (𝑜, wasSubmittedVof) ∣ ≠ 0

∣ (𝑜, wasReviewedOof−1) ∣ ≤ 2 ∣ (𝑜, wasDeterminedOof−1) ∣ = 0.

1. 𝑎𝑙𝑙𝑜𝑤 (𝑎𝑢, determine, 𝑜) ⇒ 𝑎𝑢 ∈ (𝑜, 𝑤𝑎𝑠𝑅𝑒𝑣𝑖𝑒𝑤𝑒𝑑𝐵𝑦) ∧

∣ (𝑜, wasDeterminedOof−1) ∣ = 0.

### Case’s json format

* + 1. “name” - The name of the system. Used to identify the system.
    2. “dependencyList” – An object that contains a hash map of dependencies. Each inner object (key, value) is represented by (the name of the dependency, the dependency itself). For example:

"wasOneOfReviewOof": {

"dependencyPath": [{

"dependencyPath": {

"name": "wasRevisedVof",

"isReverse": false

}

},

"ZERO\_OR\_MORE",

"ANY\_OBJECT",

{

"dependencyRole": {

"name": "review",

"depandencyName": "GENERATE"

}

},

"ANY\_OBJECT",

{

"dependencyRole": {

"name": "input",

"depandencyName": "USE"

}

}],

"isReverse": false

}

Dependency name is "wasOneOfReviewOof ", it’s not in a reverse order and the dependency path is a list that may contain a dependencyPath, a dependencyRole and operators.

Each dependencyRole has a name and a dependency name, and each dependencyPath has a name (Should be configured in the dependency list). The operator can be one of the above: OR "|", GROUP\_START "(", GROUP\_END ")", ZERO\_OR\_MORE "\*", ZERO\_OR\_ONE "?", ONE\_OR\_MORE "+", ANY\_OBJECT ".".

* + 1. “actionTypes” – An object that contains a hash map of action types. Each inner object (key, value) is represented by (the name of the action type, the action type properties). For example:

"review": {

"roles": [{

"name": "input",

"isInput": true

},

{

"name": "review",

"isInput": false

}],

"policy": {

"UARules": {

….

},

"AVRules": {

….

}

},

"isIncreaseObjetVersion": false

}

Action type name is “review”, and each action type has 3 properties:

1. “roles” – A list of related object roles, where each object role has a name and a flag if it’s an input or an output object.
2. “policy” – A policy object may contain "UARules" (User authorization rule) and "AVRules" (Action validation rule).
   1. "UARules" – Contains the policy’s User authorization rules’ description and properties.
      1. “rules” – A list of rules, which contains a specific “rule” or another "UARules".
         1. “rule” – A description of a specific User authorization rule.
            1. “operator” – May be“INCLUDE” or “NOT\_INCLUDE”
            2. “dependency” – A description of the dependency. Can be defined by the name of the pre-defined dependency path on dependency list or by a new dependency list.
      2. “operaor” – An operator to combine between all the rules. May be “AND” or “OR”

For example:

"UARules": {

"rules": [{

"rule": {

"operator": "NOT\_INCLUDE",

"depencency": {

"dependencyPath": {

"name": "wasAuthoredBy",

"isReverse": false

}

}

}

},

{

"UARules": {

"rules": [{

"rule": {

"operator": "NOT\_INCLUDE",

"depencency": {

"dependencyPath": {

"name": "wasReviewedBy",

"isReverse": false

}

}

}

}]

}

}],

"operator": "AND"

}

1. "AVRules" - Contains the policy’s Action validation rules description and properties.
   * 1. “rules” – The list of rules. It contains a specific “rule” or another "AVRules ".
        1. “rule” – A description of a specific action validation rule. May be one of theformats:
           1. Number comparable rule –

“operator” – May be “EQUAL” / ”NOT\_EQUAL” / ”BIG” / ”BIG\_EQUAL” / ”SMALL” / ”SMALL\_EQUAL”

“number” – a comparable number

“dependency” – A description of the dependency. May be defined by the name of the pre-defined dependency path on dependency list or by a new dependency list.

* + - * 1. Another object comparable-

“operator” – May be “EQUAL” / “NOT\_EQUAL” / “CONTAINED”

“leftDepencency” – A description of the left side dependency. May be defined by the name of pre-defined dependency path on dependency list or by a new dependency list.

“leftRelevantObjectRole” – A description of the object role for the input to the left dependency path.

“rigthDepencency” – A description of the right-side dependency. May be defined by the name of pre-defined dependency path on dependency list or by a new dependency list.

“rigthRelevantObjectRole” – A description of the object role for yjr input to the right dependency path.

Example:

"rule": {

"operator": "EQUAL",

"leftDepencency": {

"dependencyPath": {

"name": "wasGradedOof",

"isReverse": false

}

},

"leftRelevantObjectRole": "src",

"rigthDepencency": {

"dependencyPath": {

"name": "wasOneOfReviewOof",

"isReverse": false

}

},

"rigthRelevantObjectRole": "ref"

}

* + 1. “operaor” – An operator to combine between all the rules. May be “AND” or “OR”

For example:

"AVRules": {

"rules": [{

"rule": {

"operator": "EQUAL",

"number": 0,

"depencency": {

"dependencyPath": [{

"dependencyPath": {

"name": "wasOneOfReviewOof",

"isReverse": false

}

},

"ANY\_OBJECT",

{

"dependencyPath": {

"name": "wasGradedOof",

"isReverse": true

}

}],

"isReverse": false

}

}

}]

}

"AVRules": {

"rules": [{

"rule": {

"operator": "BIG\_EQUAL",

"number": 2,

"depencency": {

"dependencyPath": {

"name": "wasReviewedOof",

"isReverse": true

}

}

}

},

{

"rule": {

"operator": "EQUAL",

"number": 0,

"depencency": {

"dependencyPath": {

"name": "wasGradedOof",

"isReverse": true

}

}

}

}],

"operator": "AND"

}

1. "isIncreaseObjetVersion" – Whether the output object should increase the object version or create a new object id. For example:

"upload": {

"roles": [{

"name": "upload",

"isInput": false

}],

"policy": {},

"isIncreaseObjetVersion": false

}

# Project Technology

## Reliability

The project has been tested for all the end events. All the data is being loaded into the memory for every new execution.

## Availability

The system will be available until the user terminates the software.

## Maintenance

The project is developed in the most recent technology, which has a lot of documentation and is considered easy to maintain.

## Usable

The client side can be accessible from every device that includes web browser, and the server side can run on every device with Java runtime environment 1.8 installed.

## Software Dependents and IDE

**Server:**

* Written in Java 1.8
* Spring Framework – Enables MVC server side.
* Google Gson – Provides easy json library.
* Apache file upload - For uploading files from the client.
* Maven - Management and comprehension tool.
* Eclipse – IDE

**Client:**

* Written in Angular 6 (TypeScript), CSS3, HTML
* Bootstrap – A front-end component library
* Node 8.11.3
* NPM 5.6.0
* Visual Studio Code - IDE

## Running the Project

The code (server and client projects) is stored at GitHub:

<https://github.com/guybmayor/PBAC-Server>

<https://github.com/guybmayor/PBAC-Client>

In order to run the server (with pre-built client code), a project clone is needed:

*mkdir pbac-server*

*git clone* [*https://github.com/guybmayor/PBAC-Server*](https://github.com/guybmayor/PBAC-Server) *pbac-server*

Then build (make sure to the JAVA\_HOME environment variable to the JDK root):

*cd pbac-server*

*mvn install*

And run with:

*java -jar target\PBAC-0.0.1-SNAPSHOT.jar*

In order to run the client, a project clone is needed:

*mkdir pbac-client*

*git clone* <https://github.com/guybmayor/PBAC-Client> *pbac-client*

Then build (make sure you have Angular CLI installed globally):

*cd pbac-client*

*npm install*

*ng build*

Or run (will be available on http://localhost:4200/):

*ng serve*

# Conclusions

In this paper, we have described the model for provenance-based access control, and then further described the implementation process of that utilized object dependencies, which are computed based on transactions to evaluate the access requests. The model introduced a novel approach for policy specification and access evaluation by utilizing abstracted dependency names and matching dependency path patterns that are expressed using regular expressions. We showed that although the proposed model allows highly expressive policy specification, it provides a simple and effective access control management.

# References

1. By James Cheney, Laura Chiticariu, Wang-Chiew Tan, “Provenance in Databases: Why, How, and Where”, Foundations and Trends in Databases Vol. 1, No. 4 (June 2009).
2. Dang Nguyen, “PROVENANCE-BASED ACCESS CONTROL MODELS APPROVED”, The university of Texas at San Antonio (August 2014).
3. J. Park, D. Nguyen, R. Sandhu, “A Provenance-based Access Control Model”, Published in: Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on Privacy, Security and Trust.
4. Andrew Martin, John Lyle, Cornelius Namilkuo, “Provenance as a Security Control”, Proceedings of TaPP'12: the 4th USENIX Workshop on the Theory and Practice of Provenance (2012).
5. Boris Glavic , “A Primer on Database Provenance”, IIT DB Group Technical Report IIT/CS-DB-2014-01 (September 2014).
6. Julien Lacroix, Omar Boucelma, “Trusting the Cloud: a PROV + RBAC Approach”, 2014 IEEE International Conference on Cloud Computing.
7. Melanie Herschel, Ralf Diestelkämper, Houssem Ben Lahmar, “A survey on provenance: What for? What form? What from?”, The VLDB Journal (2017).

# Appendices

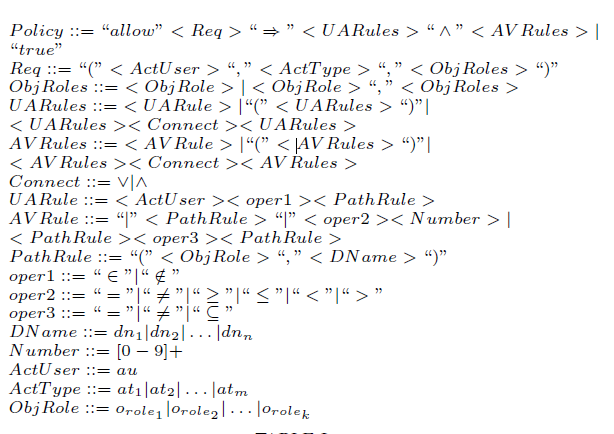
## Appendix A: COMPONENTS SPECIFICATION GRAMMAR

Based on the core components, it is possible to formally define a base model for PBAC as follows.

1. AU, 𝐴, AT, 𝑂 and 𝑂𝑅 are acting users, action instances, action types, objects, and object roles respectively.
2. 𝐺, 𝑈, G-1 and 𝑈-1 are sets of role-specific variations of ‘wasGeneratedBy’ and ‘used’ dependencies and matching sets of inverse dependencies, respectively.
3. {‘𝑐’, ‘𝑐-1’} is the set of ‘wasControlledBy’ dependency and its inverse dependency.
4. Base provenance data 𝑃𝐷B forms a directed graph and is formally denoted as a triple <𝑉𝐵, 𝐸𝐵, 𝐷𝐵 >:
5. 𝑉B = AU ∪ 𝐴 ∪ 𝑂, a finite set of acting users, action instances, and objects that have been involved in transactions in the system and are represented as vertices;
6. 𝐷B = {‘𝑐’} ∪𝑈 ∪𝐺∪{‘𝑐-1’} ∪𝑈-1∪𝐺-1, a finite set of base dependency types;
7. 𝐸**B** ⊆ {(𝐴×AU × ‘𝑐’) ∪(𝐴×𝑂×𝑈) ∪(𝑂×𝐴×𝐺) ∪ (AU×𝐴× ‘𝑐-1’) ∪(𝑂×𝐴×𝑈-1) ∪(𝐴×𝑂×𝐺-1)}, denoting dependency edges, is the set of existing base dependencies in the provenance data.
8. 𝐷𝑁O, disjoint from 𝐷B, is a finite set of abstracted names for dependencies of objects.
9. Let Σ be an alphabet of terms in 𝐷B ∪ 𝐷𝑁O. The set 𝐷𝑃𝐴𝑇𝐻 of regular expressions is inductively defined as follows:
10. ∀𝑝 ∈ Σ, 𝑝 ∈ 𝐷𝑃𝐴𝑇𝐻; 𝜖 ∈ 𝐷𝑃𝐴𝑇𝐻;
11. (𝑃1∣𝑃2), (𝑃1. 𝑃2), 𝑃1∗, 𝑃1+, 𝑃1? ∈ 𝐷𝑃𝐴𝑇𝐻, where 𝑃1 ∈ 𝐷𝑃𝐴𝑇𝐻 and 𝑃2 ∈ 𝐷𝑃𝐴𝑇𝐻.
12. 𝐷𝑃𝐴𝑇𝐻B ⊆ 𝐷𝑃, is the set of regular expression using only alphabet of terms in 𝐷B.
13. DLO: 𝐷𝑁O → 𝐷𝑃𝐴𝑇𝐻, defines each 𝑑𝑛 ∈ 𝐷𝑁O as a path expression. 𝐷𝐿O is also viewed as a list of pairs of object dependency names and corresponding dependency paths.
14. O: 𝐷𝑁O → 𝐷𝑃𝐴𝑇𝐻B, maps each 𝑑𝑛 ∈ 𝐷𝑁O to a path expression using only base dependency types 𝑑𝑏 ∈ 𝐷**B** by repeatedly expanding the definitions of any 𝑑𝑛𝑖 ∈ 𝐷𝑁𝑂 that occurs in 𝐷𝐿O(𝑑𝑛).
15. 𝑃𝐸 is a language specified in the policy expression grammar 𝑃𝐺.
16. 𝑃 ⊆ PE, is a finite set of policies.
17. : 𝐴𝑇 → 𝑃, a mapping of an action type to a policy.
18. 𝛿O: 𝑂 × 𝐷𝑃𝐴𝑇𝐻B → 2VB, a function mapping an object and a base dependency path to vertices in 𝑃𝐷**B** such that 𝑜2 ∈ 𝛿 (𝑜1, 𝑑𝑝𝑎𝑡ℎ) if there exists a path in 𝑃𝐷B from 𝑜1 to 𝑜2 whose edge labels form a string that satisfies the regular expression 𝑑𝑝𝑎𝑡ℎ.

The simplicity and effectiveness in policy specification and access control management are achieved with the utilization of dependency names and matching dependency paths in dependency list (DLO) as shown in Definition 5-9. Definition10-12 provides the means for defining policies and attaching them to action types. Definition 13 defines the 𝛿O function necessary for access request evaluation with respect to the given 𝑃𝐷B.

## Appendix B: POLICY SPECIFICATION GRAMMAR



## Appendix C: 𝐴𝑐𝑐𝑒𝑠𝑠𝐸𝑣𝑎𝑙𝑢𝑎𝑡𝑖𝑜𝑛(𝑎𝑢, 𝑎,𝑂) ALGORITHM

