



**University of Hamburg**  
Department of Informatics  
Security in Distributed Systems  
Sicherheit in Verteilten Systemen (SVS)



# **Using the Jess Rule Engine to Evaluate Authorization Policies**

Baccalaureate Thesis • Baccalaureatsarbeit

**Bodo Eggert**

Advisor:  
Prof. Dr.rer.nat. Joachim Posegga  
Technical advisor:  
Christopher Alm

Hamburg, September 26, 2007



# Contents

<b>1 Introduction</b>	<b>4</b>
1.1 Context / Overview . . . . .	4
1.2 Goal . . . . .	4
1.3 Gain . . . . .	4
1.4 Organization of the Thesis . . . . .	4
<b>2 The Jess Rule Engine</b>	<b>5</b>
2.1 Role Engines - Terms and Definitions . . . . .	5
2.2 The Jess Language . . . . .	6
<b>3 RBAC</b>	<b>11</b>
3.1 Basic RBAC Definitions . . . . .	11
3.2 Core RBAC . . . . .	11
3.3 Hierarchal RBAC . . . . .	12
3.4 Constrained RBAC . . . . .	13
<b>4 Design and Implementation</b>	<b>14</b>
4.1 My Variation of the Core RBAC Model: . . . . .	14
4.2 Request Handling in a Rule Engine . . . . .	15
4.3 RBAC Data Model . . . . .	15
4.4 Operations on the RBAC Data . . . . .	18
<b>A Source Code</b>	<b>23</b>
<b>B Example Session</b>	<b>27</b>
<b>References</b>	<b>36</b>

# **1 Introduction**

## **1.1 Context / Overview**

A rule engine is a tool for processing a dynamic set of facts according to a set of rules. Each change in the set of facts - including the initial definition - may activate some rules. One activated rule will fire and perform the assigned action, possibly changing the set of facts again.

Rule engines may be used to process events; each event will be represented using a new fact, and by carefully designing the rule set, each event will eventually trigger an action.

Using a rule engine in a rule-based authorization system, the incoming events are access requests that need to be answered by grant/deny access decisions. If, for example, a user asks for access to a certain file, a corresponding fact will be generated and a certain rule matching this request may specify that he or she is granted access.

One example of rule engines is the *Jess Rule Engine* [Jesa] and scripting environment. Jess has been written entirely in Sun's Java language by Ernest Friedman-Hill at Sandia National Laboratories in Livermore, CA. It provides an efficient rule engine and the capability of interfacing Java applications.

## **1.2 Goal**

The goal of this thesis is to figure out how and to what extent the Jess rule engine can be used as a decision engine for access control systems. In particular, practical implementation examples are given showing how an access control system could be realized using Jess. This thesis does not examine interfacing the rule engine from an application.

## **1.3 Gain**

The work of this thesis can be used as a basis for an evaluation of Jess - and similar rule engines - during the development of an authorization engine. In particular, a set of rule language constructs that are necessary to form authorization rules will be revealed which gives insight into the properties.

## **1.4 Organization of the Thesis**

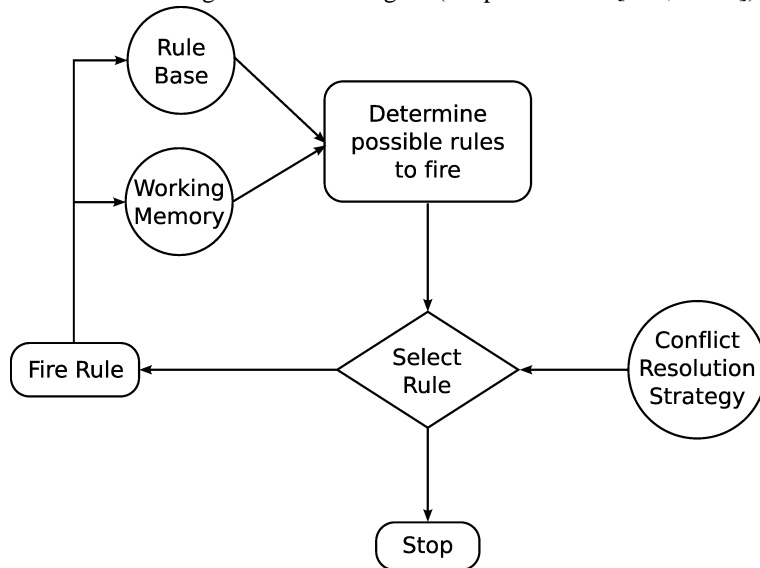
This thesis is organized in three main parts. In chapter 2, I describe the concept of rule engines and, as far as it is needed to understand this thesis, the syntax and the semantics of the Jess language. Chapter 3 gives an overview of the RBAC definitions from the American National Standard. My implementation of the RBAC decision engine in Jess is discussed in Chapter 4.

## 2 The Jess Rule Engine

### 2.1 Role Engines - Terms and Definitions

A rule engine is an engine, which matches a set of facts against a set of rules (called *working memory*), and performs an action for matching rules.

Figure 1: A rule engine (simplified from [Dro, Ch. 1])



A *fact* is much like a database record, it consists of a number of named *slots*, which would be stored in the columns of a table. There is no explicit grouping of facts into tables, but each fact will be structured like its template, and any match will use that template name to select a class of facts, similar to the table name in select clauses. *asserting* a fact will create this fact, *retracting* a fact will remove the fact from the working memory.

Facts are structured using a *template*, which corresponds to the structure of a table. It names a kind of fact, and it tells about the number of slots, their names and possibly their type. Templates can inherit their properties from other templates, making all rules which apply to the original type of fact to the inheriting type of fact, too.

A *rule* is a mapping from one list of conditions to one list of actions. Each *conditional element* from the list of conditions is matched against all possible facts, and iff all elements of the list do have an assigned fact<sup>1</sup>, the rule is *active*. The engine will choose one of the active rules and process its actions. This can (and usually will) change the defined set of facts, causing other rules to be activated or deactivated. If there are no more active rules, the engine may halt automatically or wait for the set of facts to change by external means.

If there are active rules, one of them is selected and its corresponding list of actions is executed - the rule *fires*. Rules are not fired in a specific order. If the rule engine is

<sup>1</sup>Some special Conditional Elements do not require a matching fact

busy, there is no guarantee that a rule matching a given fact will ever fire, as long as there are other rules which might fire instead. However, rules do have a property named “salience”, which effectively groups them into priority classes. If one rule matches, no rule with a lower salience value will fire. This cannot be used to guarantee an order of execution for similar facts, but it can be used to guarantee important events like e.g. changes in the permission set to be processed before file access.

It is possible to ensure a sequential evaluation by adding a counter to each fact and only matching facts if this counter has the next number. If you do that, you should not be using a rule engine in the first place.

## 2.2 The Jess Language

The Jess syntax is very similar to LISP, but it does behave differently by not using cons cells, allowing nested lists for data nor providing macros or user-defined special forms (lazy evaluation).

Like in LISP, each statement is a value, a symbol or a list. Values or symbols evaluate to themselves, and lists are evaluated by calling the first element of the list as a function, passing the rest of the list as arguments. Unlike LISP, the empty list is not equal to nil.

### Templates

```
(deftemplate template-name
  ["Documentation comment"]
  [(declare (slot-specific TRUE | FALSE)
            (backchain-reactive TRUE | FALSE)
            (from-class class name)
            (include-variables TRUE | FALSE)
            (ordered TRUE | FALSE))]
  (slot | multislot slot-name
   [(type ANY | INTEGER | FLOAT | NUMBER
        | SYMBOL | STRING | LEXEME | OBJECT | LONG)]
   [(default default value)]
   [(default-dynamic default value)])*)
```

**template-name** is the name of the template.

**declare** starts the block of declarations:

**slot-specific** Usually, if one fact is modified, each rule matching that fact is activated. Using slot-specific, it is only activated if it matches the specific slots. This can be used if one rule is used to update one slot of a fact whenever one of the other slots change.

**backchain-reactive** If a rule is backchain-reactive, a special goal seeker template named “need-”+template-name is created, and each time a corresponding but not yet defined fact may be needed, a “need-”-fact is asserted. This is supposed to be used in order to create a rule which will assert a fact whenever it’s needed.

**from-class** from-class is used to create templates from Java objects. For each getFoo method, a slot named foo will be created.

**include-variables** If a template is created from a Java class and if include-variables is true, Jess will additionally create one slot for each public member variable.

**ordered** Ordered facts do not contain slots, but they are a list of values. Internally, this is done by using exactly one multislot<sup>2</sup> and hiding it's name. All facts not corresponding to defined templates are ordered facts.

**slot** A slot is a named storage for a simple value. A slot can be restricted to one type of values.

**multislot** A multislot is a named storage for lists of simple values. A multi-slot can be restricted to one type of values.

#### **deffacts - define facts.**

```
(deffacts deffacts-name
 [ "Documentation comment" ]
 (fact)* )
```

Facts defined using deffacts are automatically asserted on each engine reset, while facts asserting using (assert) are discarded. Deffacts does not immediately assert the facts.

#### **deffunction - define function**

```
(deffunction function-name (argument*)
 [ "Documentation comment" ]
 (function call)* )
```

Defined functions can be called like built-in functions. Each parameter is bound to exactly one argument variable. Placing a dollar sign before the variable in the argument list causes this variable to be bound to a list of parameters. It is not possible to pass lists of values for any but the last argument variable, since Jess does not support nested lists.

#### **defglobal - define global variables**

```
(defglobal [?*name = value]+ )
```

defglobal defines a global variable and sets it's initial value. The name of a global variable must begin and end with “\*”. Each variable is prefixed using “?”, like each PHP variable is prefixed using “\$”.

---

<sup>2</sup>named \_\_data

### **defquery - define a query**

```
(defquery query-name
  ["Documentation comment"]
  [(declare (variables variable+)
    (node-index-hash value)
    (max-background-rules value))]
  (conditional element)* )
```

Upon invocation, a query matches the set of facts and returns a list of facts matching the list of conditional elements. A query can be invoked using the `run-query*` function or the `count-query-results` function. The facts from the result of `run-query` can be extracted using e.g.

```
(while (?result next)
  (printout t (?result getString first-name)
    " "      (?result getString last-name)
    ", age " (?result getInt age) crlf))
```

The semantics of a query definition are:

**query-name** The name of the query.

**declare** starts the block of declarations:

**variables** This list of variables corresponds to the argument variables of functions

**node-index-hash** Used to tune the hash table size. Should be a prime number.

**max-background-rules** The number of rules that may fire in order to allow backward chaining.

### **defrule - define a rule**

```
(defrule rule-name
  ["Documentation comment"]
  [(declare (salience value)
    (node-index-hash value)
    (auto-focus TRUE | FALSE)
    (no-loop TRUE | FALSE))]
  (conditional element)*
  =>
  (function call)* )
```

**rule-name** The name of the rule

**declare** starts the block of declarations:

**salience** The salience value determines the order, in which rules will fire. If there are rules having a higher salience than a given rule, this rule will not fire.

**node-index-hash** Used to tune the hash table size. Should be a prime number.

**auto-focus** Automatically change the focus to the defining module. Modules and focus are not described in this document.

**no-loop** Even if this rule modifies facts, this does not cause this rule to fire again immediately.

=> A special symbol separating the left hand side of this symbol, the conditional elements, from the right hand side, the list of function calls.

### conditional elements

Conditional elements are patterns matching one fact each. They are used to select by their properties and to join facts on common properties. Additional constraints may be imposed on

[ ?variable <- ] (template-name pattern|test\* ) | (special-conditional-element ...)

**?variable** <- binds the matching fact to the variable. This variable can be used to access (or delete) this fact.

**template-name** Facts conforming to this template are matched against this conditional element

**slot-pattern** A tuple of a slot name, followed by a number of values, variables, function calls or regular expressions, combined using '&', '|' or '~', for logical 'and', 'or' and 'not'. The named slot will be compared to each element.

**(nothing)** Not specifying anything matches an empty list from a multislot

**immediate values** match if the slot contains this value.

**?** A question mark matches any value. **\$?** will match any list.

**?variables** matches the value of this variable, or binds the value from this slot to this variable (if the variable was undefined). After a variable is bound, it can be used as a parameter for function calls. If the variable was prefixed using a dollar sign, it matches a list of values. Common variables bound slots from multiple conditional elements result in a join of the corresponding facts on these slots.

Variable names are prefixed using "?".

**Function calls** can be started using an equals sign or a colon. Using an equal sign, they match their return value, while using a colon, they match if they return TRUE.

**Regular expressions** will match as expected.

**logic equation** An expression surrounded by braces. Within this expression, the following elements may be combined:

**immediate value**

**slot name** the value of this slot

**<** less than

<= less than or equal to  
> greater than  
<= greater than or equal to  
== equals  
!= not equal to  
<> not equal to, alternate syntax  
&& and  
|| or

**Special-conditional-elements** are used to group other conditional elements or to perform some tests independently from corresponding facts.

**and** and-combines a list of conditional elements

**or** or-combines a list of conditional elements

**not** negates one conditional element. Variables defined within that element are scoped within the not element, for obvious reasons.

**exists** makes one conditional expression match only for one fact, even if there are multiple facts that would match. exists(...) is equal to not(not(...)).

**test** calls one function and matches if the function returns true. Using test should be avoided, unless the result of this function may change.

**logical** makes one conditional element be the logical support of each fact this rule will assert, binding it's lifetime to the validity of the logically supporting match. Multiple matches may independently assert this fact, and it can be unconditionally asserted as well. Facts aren't automatically removed unless all logically supporting facts are retracted.

**forall** matches only if for each possible match in the group, all other conditional elements do match, too.

**accumulate** This is not a simple match, but a loop over the facts set performing some program logic. Please refer to the Jess documentation [Jesb] for a detailed description.

**unique** deprecated and ignored

## 3 RBAC

In companies, access rights to objects are usually granted by the role a user plays. A dishwasher has access to soap, and a CEO will have access to his office. If the CEO is promoted to dish washer, he will lose access to his office and gain access to the soap.

RBAC tries to apply this model to computing in order to protect each individual resource. Each user is assigned to a set of defined roles, and depending on the role he plays currently, access will or will not be granted. Since this is a mandatory access control model, no user can grant his access privileges to other users.

RBAC, as defined in [RBA04] is divided into a set of modules, which may be implemented depending on the specific needs.

### 3.1 Basic RBAC Definitions

RBAC defines these terms for use within the standard:

**Object** An object is any resource subject to user access control, e.g. a file, a printer or a database record.

**Operations** According to the standard, an operation is an executable image of a program (a process). Examples given are the file operations read, write and execute and the database operations insert, delete, append and update.

**Permission** Permissions are approvals to perform an operation on an object.

**User** A user is a person or, using the extended definition, a machine, a service, a network or an intelligent autonomous agent.

**Role** A Role is a job function performed by a user. Roles differ in authority and responsibility conferred on users.  
Each user has at least one role, and in each Session, he will choose a subset of these roles.

**Session** A session is a mapping between users and their activated subset of roles.  
When logging in to a system and starting a session, a user will choose a subset of his allowed roles. (The combinations may be further restricted by the implementation.)

### 3.2 Core RBAC

Core RBAC is, as the name suggests, the minimum part, which must be implemented by each RBAC system. It includes the above five sets of elements, and a set of relations including:

**UA** User Assignment: A set of users and their assigned (=possible) roles  
 $UA \subseteq \text{Users} \times \text{Roles}$

**PRMS** Permissions: The set of Operations and Objects.  
 $PRMS = 2^{\text{Operations} \times \text{Objects}}$

**PA** Permission Assignments: The set of Permission-to-Role-assignments.

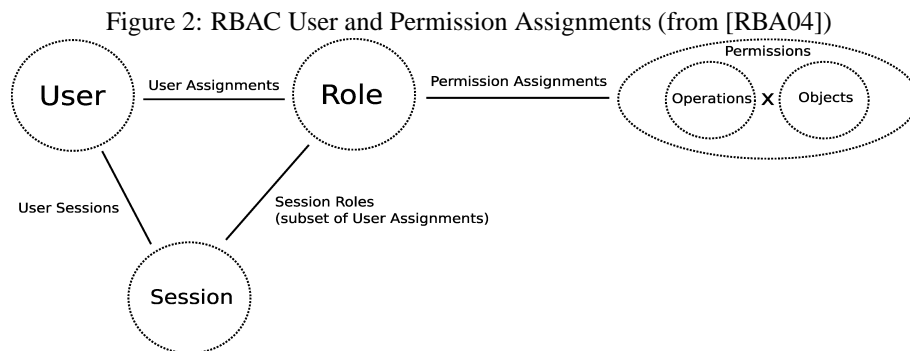
$$PA \subseteq PRMS \times Roles$$

**SESSIONS:** The set of Sessions. While no further details are given at the definition, later parts assume it to be a set of Names.

It also defines several relations, of which most duplicate the content of the above sets. Only the session relations are needed:

$session\_users(s : SESSIONS) \rightarrow USERS$ , the mapping of session  $s$  onto the corresponding user

$session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$ , the mapping of session  $s$  onto the active set of roles for this session



### 3.3 Hierarchal RBAC

Hierarchal RBAC (as all other modules) is an extension to Core RBAC, which provides a hierarchal relation between roles:

**Inheritance** If all permissions of one role  $r_1$  are also granted to the role  $r_2$ ,

$$r_2 \text{ inherits } r_1.$$

**Containment** If all users authorized for  $r_1$  are also authorized for  $r_2$ ,

$$r_1 \text{ contains } r_2.$$

According to this definition, the definition of users being assigned to a role is redefined<sup>3</sup> to the set of users of that role or any role inheriting from that role, and the set of permissions assigned to a role is redefined to the permissions assigned to that role or to a role containing that role.

Later parts of the standard do not use this definition, but assume:

**Inheritance** If  $r_2$  inherits  $r_1$ , all permissions of role  $r_1$  are also granted to the role  $r_2$ ,

**Containment** If  $r_1$  contains  $r_2$ , all users authorized for  $r_1$  are also authorized for  $r_2$ .

<sup>3</sup>with respect to the Core RBAC definition

*Limited Role Hierarchies* do impose the restriction

$$\forall r, r_1, r_2 \in \text{Roles}, r \succeq r_1 \wedge r \succeq r_2 \rightarrow r_1 = r_2$$

or

$$\forall r, r_1, r_2 \in \text{Roles}, r \preceq r_1 \wedge r \preceq r_2 \rightarrow r_1 = r_2$$

Limited Role Hierarchies do not provide multiple inheritances.

### **3.4 Constrained RBAC**

Constrained RBAC adds Separation of Duty relations. Separation of Duty is used to ensure that fraud or major errors do not occur without coercion of multiple users by assigning different individuals to separate tasks required in the performance of a business function.

There are two kinds of Constrained RBAC, Static and Dynamic Separation of Duty (SSD and DSD). SSD is done by ensuring that if a user is assigned a role, the user must not be assigned to a conflicting role. There are possible less restrictive Static SSD models, e.g. allowing users assignments to m of n roles necessary for a task.

DSD differs from SSD in not limiting the possible set of user assignments, but the set of roles a user may select at a given time.

## 4 Design and Implementation

In this chapter, I describe the design and implementation of an RBAC engine based on Jess.

I will implement a variation of the Core RBAC module, which shall be equivalent in functionality, but using different internal data representation. I will explain why these data structures are equally capable of representing the same relations.

### 4.1 My Variation of the Core RBAC Model:

**UA** User Assignment: A set of users and their assigned (=possible) roles

$$UA \subseteq \text{Users} \times \text{Roles}$$

**PRMS** Permissions: The set of Operations and Object Types.

$$PRMS = 2^{\text{Operations} \times \text{Objects}}$$

**SESSION:** The set of active user sessions. Each session is assigned to one user and a subset of his roles.

$$\text{Session} = \{session\_id, user, roles\} \subseteq \mathbb{N} \times \text{Users} \times 2^{\text{Roles}}$$

where  $\text{Roles} = \{role : (user, role) \in UA\}$  and  $session\_id$  is a unique ID.

**PA** Permission Assignments: The set of Permission-to-Role-assignments.

$$PA \subseteq \text{Operations} \times \text{Objects} \times \text{Roles}$$

This definition differs from the original definition in:

The permission Assignment uses Object Types, while the original definition requires an assignment for each individual object. By defining one Object Type per object, this implementation behaves like the original definition, but defining assignments to similar objects is eased by using Object Types.

The original definition of *Permission Assignments*,  $PA \subseteq PRMS \times \text{Roles}$ , allows different representations of the same permission, e.g. the tuple  $(\{(read, file), (write, file)\}, role)$  can be expressed as two tuples:  $(\{(read, file)\}, role), (\{(write, file)\}, role)$ . More generally, you can show a distributive law:

$((perm_1 \cup perm_2), role) \Leftrightarrow (perm_1, role) \cup (perm_2, role)$ . Using this scheme<sup>4</sup>, you can easily show that limiting PRMS to  $\text{Operations} \times \text{Objects}$  or changing PA to  $PA \subseteq \text{Operations} \times \text{Objects} \times \text{Roles}$  is - from the perspective of the engine<sup>5</sup> - equivalent.

From the perspective of a user, it may be beneficial to have pre-designed sets of permissions, which can be assigned to roles. A user interface providing this feature would e.g. use the original model and extend it by allowing to name and describe these sets. Since this user interface can easily do the transformation, there is no disadvantage from not supporting permission sets within the engine.

---

<sup>4</sup>and, off cause,  $(\emptyset, role) \Leftrightarrow \emptyset$ .

<sup>5</sup>Users might like to group common sets of permissions, e.g. all permissions necessary for accessing the network. This can be done by the user interface, which is out of the scope of this document.

The *SESSION* set is originally just a set of IDs, while starting sessions is equivalent to changing the two session relations. I combine these relations into the set of sessions, which gives me the convenience of manipulating only one object while at the same time using and exploring Jess' capability of handling lists.

Another possible model would use  $\text{Session} = \{session\_id, user, roles\} \subseteq \mathbb{N} \times \text{Users} \times \text{Roles}$  instead of  $\dots \times 2^{\text{Roles}}$ , creating multiple facts for sessions if they have more than one active role. This model would either need a special Null-role for sessions without active roles, or it would need to ignore these sessions. Either way, the default-deny-policy will block all requests for these sessions. The benefit of this model is using tables conforming to the first normal form, enabling easier pattern matching and presumably being faster.

## 4.2 Request Handling in a Rule Engine

Processing a request using a rule engine consists of three main steps:

1. Pass requests into the rule engine
2. Let the rule engine allow or deny the request
3. Pass the result to the system

I do not need to implement 1. and 3. myself, since my work will only use the Jess language directly. However, considering a possibility of further use, I will use a way which allows Java to pass request objects to Jess. Jess can generate facts from Java objects using `Jess.Rete.add()` and calling methods of these objects. (For details, please refer to the Jess manual [Jesb])

Within a traditional system, requests like reading a file or creating a session are either processed synchronously, or they are enqueued within one or a number of queues. A rule engine does not provide queues, nor is it designed to process requests directly. Instead, it uses a set of facts and matches it against a set of rules - either on start of the engine or if the set of rules changes during runtime. As stated before, this does not guarantee any fairness, events can starve forever.

## 4.3 RBAC Data Model

### User Assignment and Permission Assignment data

The *User Assignment* is a m:n relation, which is naturally implemented by asserting a UA fact for each assigned pair of user-id and role. The slots are named

**uid** The ID of the User, who is being assigned

**role.** The role to which this user is assigned

*Permission Assignments* are implemented twice, once using only role and the object types, and once using role, intention<sup>6</sup> of the access and the object types. The former kind of Permission Assignment is useful when there are no restrictions imposed on the intention, while the later is useful for restricted access to an object, e.g. allowing only to read /etc/resolv.conf.

For each of the implementations, there is one template: *grant-by-role* and *grant-by-role-intention*. These templates contain the slots:

**urole** The role to which this permission was assigned.

**intent** The intent which was granted on the object. For *grant-by-role*, this slot does not exist.

**objtype** The type of object which was accessed.

```
(deftemplate UA
"The User Assignment m:n relation"
(slot uid)
(slot role))
```

```
(deftemplate grant-by-role
"A simplified Permission Assignment,
disregarding the intent"
(slot urole)
(slot objtype))
```

```
(deftemplate grant-by-role-intent
"The Permission Assignment m:n:o relation"
(slot urole)
(slot intent)
(slot objtype))
```

## Sessions

Each active user has at least one open session, usually starting with logging in and ending when the last program of the session terminates. In RBAC systems, users must chose a set of active roles from their assigned roles, possibly restricted by Dynamic Separation of Duty.

The user Sessions are modeled using a Session fact, which consists of the slots

**id** The Session-ID: This system-unique number identifies the session

**uid** The User-ID: This slot identifies the user who initiated this session

---

<sup>6</sup>Intention is e.g. reading or writing a file, printing on a printer, canceling a print job, ...

**roles** The set of active roles for this session

Since the decision logic is supposed to reside in the rule engine, I need to pass the information about started sessions into the rule engine. I might have used normal Session facts plus one flag indicating valid sessions, but this would add overhead to valid sessions and enlarge the set of patterns for the rules for creating sessions. Instead, I chose to define a separate template CreateSession having the slots

**id** The desired session id, assumed to be unique

**uid** The identity of the user

**roles** The set of requested roles

**groles** Internal use, the list of roles granted so far

**result** The result of the rule evaluation

The return slot will be set to the system response in order to see the outcome of evaluating a specific rule. The rules are designed to only react to facts if the result has not yet been set. If the event fact would be generated from a program using the Jess engine, you would call an object method instead (passing the result to the originator), and this method would remove the fact from the rule engine.

This fact is first checked against the DSD restrictions, each role is checked against the User Assignment, and if there are no violations, a Session will be created and the CreateSession fact will be removed. If there is a violation, the default rule will assign "denied" to the result slot.

```
(deftemplate Session
"A Session consists of:
  id:      The Session-ID: This system-unique number
           identifies the session
  uid:     The User-ID: This slot identifies the user who
           initiated this session
  roles:   The set of active roles for this session"
(slot id)
(slot uid)
(multislot roles))

(deftemplate CreateSession
"Request for creating a session
  id:      The desired session id
  uid:     The identity of the user
  roles:   The set of requested roles
  groles:  Internal use, the list of roles granted so far
  result:  The result of the rule evaluation"
(slot id)
(slot uid))
```

```
(multislot roles)
(multislot groles)
(slot result))
```

### Access Control

Accessing a system object is implemented using the Access template. It consists of the slots

**session** The user session which requested this access

**objtype** The type of the requested object

**intent** The intent, e.g. read, write, print ...

**result** The result of the rule evaluation.

This result slot has the same purpose as the result slot above, but instead of removing this fact, the result will be set to “granted” iff there is a rule allowing this access.

```
(deftemplate Access
"Access request from a session for a file.
 session: The user session which requested this access
 objtype: The type of the requested object
 intent: The intent, e.g. read, write, print ...
 result: The result of the rule evaluation"
 (slot session)
 (slot objtype)
 (slot intent)
 (slot result))
```

### Dynamic Separation of Duty

The DSD template is defined as a multislot named roles, containing set of roles, and a corresponding integer slot  $n$ ,  $n > 1$  (not enforced). It forbids creating sessions having  $n$  or more roles from the specified set.

## 4.4 Operations on the RBAC Data

### Default deny access control rule

RBAC will deny access unless there is a corresponding Permission Assignment. This behavior is implemented using a default rule named `default_deny_Access`. This rule will match any Access fact, resulting in a conflict with the more specific rules which would possibly grant the access. The rule engine would choose one random rule - unless they have a different salience value.

The salience value of the default\_deny rules is set to -100 (default = 0), instructing Jess not to fire this rule unless there are no other active rules of higher salience.

```
(defrule default_deny_Access
"This rule will fire if there are no other rules,
providing the default-deny behaviour"
; Don't run unless there are no other rules:
(declare (salience -100))
; fire only for rules without a result:
?f <- (Access (result nil))
=>
(printout t "Permission denied" crlf)
(modify ?f (result "denied")))
```

### **Allow Access by Role**

The rule access-by-role-r will allow access if there is an applicable Permission Assignment. The PA's roles must be a member of the session's role set, and the PA's objtype must match the requested objtype.

```
(defrule access-by-role-r
"This rule allows access to an object, if there is a
Permission Assignment granting access to an object for
a member role of this session."
?f <- (Access (session ?session) (objtype ?objtype)
              (result nil))
(Session (roles $?roles) (id ?session) )
(grant-by-role (urole ?urole
               & :(member$ ?urole ?roles))
              (objtype ?objtype) )
=>
(printout t "Access Granted" crlf)
(modify ?f (result "granted")))
```

### **Allow Access by Role and Intent**

The rule access-by-role-intent-r will allow access if there is an applicable Permission Assignment. The PA's roles must be a member of the session's role set, and the PA's intent and objtype must match the requested intent and object type.

```
(defrule access-by-role-intent-r
"This rule allows access to an object, if there is a
Permission Assignment granting access to an object for
a member role of this session having the correct
intention."
```

```

?f <- (Access (session ?session) (intent ?intent)
          (objtype ?objtype) (result nil))
(Session (roles $?roles) (id ?session))
(grant-by-role-intent (urole ?urole
                      & :(member$ ?urole ?roles))
                    (intent ?intent)
                    (objtype ?objtype))

=>
(printout t "Access Granted" crlf)
(modify ?f (result "granted"))

```

### Default Deny Session Creating Rule

Creating sessions is prohibited, unless explicitly allowed. This rule will match all CreateSession facts and deny the implied request, unless one of the CreateSession facts does match, too.

```

(defrule default_deny_Session
"This rule will fire if there are no other rules,
providing the default-deny behaviour"
; Don't run unless there are no other rules:
(declare (salience -100))
; fire only for rules without a result:
?f <- (CreateSession (result nil))
=>
(printout t "I'm sorry, Dave, I can't do that" crlf)
(modify ?f (result "denied")))

```

### Initial Step of Creating Sessions

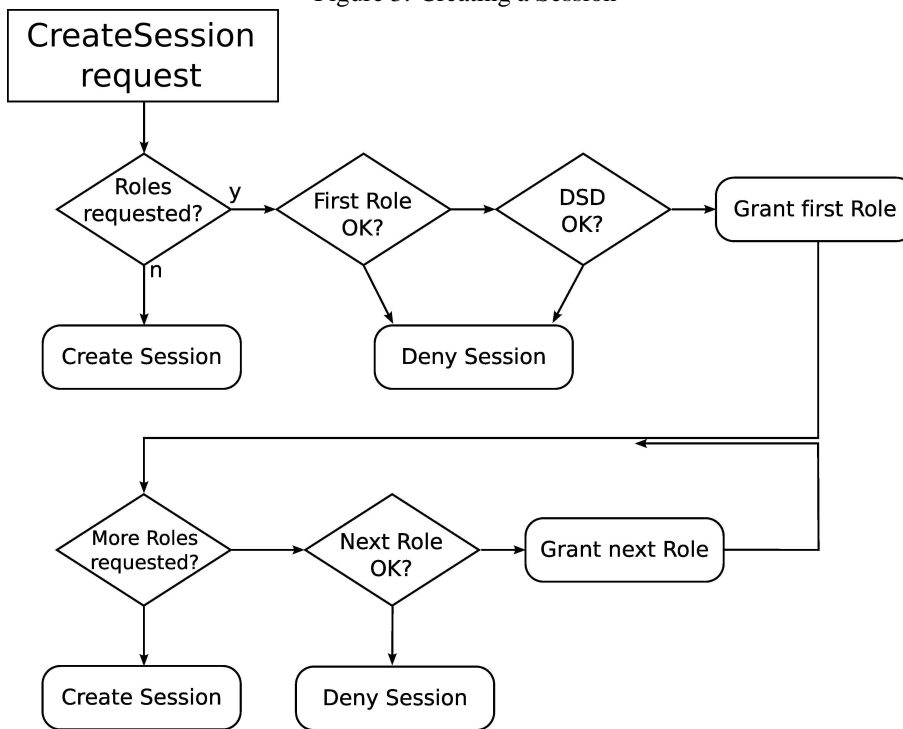
Creating sessions is done in three steps. This first rule will fire if a new CreateSession fact appears and if there are rules requested for this session. If there are no roles requested, this session will not have any RBAC privileges, and RBAC should not deny this kind of sessions.

This first rule matches the set of requested roles against the DSD restrictions and it matches the first rule of the requested set against the User Assignment. If there is a valid User Assignment and no DSD is violated, the first requested role is granted and the CreateSession fact is modified by moving the first role from the requested set in slot roles to the granted set in groles.

Since the description of DSD in the code below is hard to read, I will repeat it here:

The DSD check is quite tricky: We create a union of the rules from the request and each DSD entry in turn. While creating this union, the intersecting roles are removed from the resulting list, therefore the number of duplicates is the difference between the combined length of the original lists and the length of the union.  $|A \cap B| = |A| + |B| - |A \cup B|$

Figure 3: Creating a Session



DSD restrictions specify a number of roles  $n$ . If at least this number of roles intersect with the requested set, the request shall be denied.

Therefore we get  $|\text{requested\_roles}| + |\text{dsd\_roles}| \geq |\text{requested\_roles} \cup \text{dsd\_roles}| + n$  for all non-conforming combinations.

```
(defrule CreateSession-InitialStep
"Create a Session - First Check
```

This is the initial step, unless no roles were requested. The set of rules is checked against the DSD table, and the first role is checked against the User Assignment.

If both checks succeed, the first role is moved to the g(ranted-)roles list. This is repeated until all rules have been checked against the User Assignment.

In order to not check the DSD on each step, this rule has been copied, and the DSD check has been removed from the copy while adding a check for a (non-)empty groles list to each version."

```
?f <- (CreateSession (id ?id) (uid ?uid)
```

```

                                (roles ?role1 $?roles)
                                (groles /* empty */)
(UA (uid ?uid)
    (role ?role1))

; The DSD check is quite tricky: We create a union of
; the rules from the request and each DSD entry in turn.
; While creating this union, the intersecting roles are
; removed from the resulting list, therefore the number
; of duplicates is the difference between the combined
; length of the original lists and the length of the
; union.

; DSD restrictions specify a number of roles n. If at
; least this number of roles intersect with the
; requested set, the request shall be denied.

; Therefore we get |requested_roles| + |dsd_roles| >=
; |requested_roles (union) dsd_roles| + n
; for any non-conforming conforming combinations.

```

### Iterating Step of Creating a Session

As long as there are session requests having roles in the requested set, each role in turn will be checked against the User Assignment and - if there is an applicable assignment, be moved to the set of granted roles. This is essentially the same thing the initial step does, but this rule will only fire if the initial step rule has moved at least one rule to the granted set, which implies this request has passed the DSD check, allowing us to skip it here.

This step could also be performed by the initial rule, but this would rerun the DSD check for each rule.

```

(defrule CreateSession-NextStep
"Create a Session - Iterating Check

check the next role from the list of requested roles.
DSD checks have been done by the rule above."

; The DSD check has been done.
?f <- (CreateSession (id ?id) (uid ?uid)
        ; Need to check roles?
        (roles ?role1 $?roles)
        ; groles is non-empty?
        (groles ?grole1 $?groles)
        (result nil))

(UA (uid ?uid)
    (role ?role1))

```

```
=>
;move the granted role to groles:
(modify ?f (groles (list ?grole1 ?groles ?role1))
          (roles ?roles))
```

### Final Step of Creating a Session

The final step is run in two cases:

1. The first rule and the iterating step will move all roles from the requested set of a CreateSession request to the granted set until it's empty, and all roles in the granted roles set may be active in the session.
2. If no roles are requested, the session shall be granted. The groles slot is empty by default.

In both cases, the roles slot is empty and the groles slot contains all roles for the session. Granting a session is done by creating a Session fact, copying the session-id, the user-id and the set of granted permissions. Since all relevant informations are copied to the Session (the result is implied by having the Session fact), I may retract the CreateSession fact.

```
(defrule CreateSession-FinalStep
  "Create a Session - Instantiate Session Fact

  grant Session if all requested roles have been
  processed, or if no role was requested."
  ?f <- (CreateSession (id ?id) (uid ?uid)
                    (roles $?roles
                     & : (= (length$ ?roles) 0))
                    (groles $?groles))

=>
(assert (Session (id ?id) (uid ?uid) (roles ?groles)))
(retract ?f))
```

## A Source Code

```
(deftemplate Access
  "Access request from a session for a file.
  session: The user session which requested this access
  objtype: The type of the requested object
  intent: The intent, e.g. read, write, print ...
  result: The result of the rule evaluation"
  (slot session)
  (slot objtype))
```

```

(slot intent)
(slot result))

(deftemplate Session
"A Session consists of:
id:    The Session-ID: This system-unique number
       identifies the session
uid:   The User-ID: This slot identifies the user who
       initiated this session
roles: The set of active roles for this session"
(slot id)
(slot uid)
(multislot roles))

(deftemplate UA
"The User Assignment m:n relation"
(slot uid)
(slot role))

(deftemplate grant-by-role-intent
"The Permission Assignment m:n:o relation"
(slot urole)
(slot intent)
(slot objtype))

(deftemplate grant-by-role
"A simplified Permission Assignment,
disregarding the intent"
(slot urole)
(slot objtype))

(defrule default_deny_Access
"This rule will fire if there are no other rules,
providing the default-deny behaviour"
; Don't run unless there are no other rules:
(declare (salience -100))
; fire only for rules without a result:
?f <- (Access (result nil))
=>
(printout t "Permission denied" crlf)
(modify ?f (result "denied")))

(defrule access-by-role-r
"This rule allows access to an object, if there is a
Permission Assignment granting access to an object for
a member role of this session."
?f <- (Access (session ?session) (objtype ?objtype)
            (result nil))
(Session (roles $?roles) (id ?session) )
(grant-by-role (urole ?urole

```

```

                & :(member$ ?urole ?roles))
                (objtype ?objtype) )
=>
(printout t "Access Granted" crlf)
(modify ?f (result "granted")))

(defrule access-by-role-intent-r
"This rule allows access to an object, if there is a
Permission Assignment granting access to an object for
a member role of this session having the correct
intention."
?f <- (Access (session ?session) (intent ?intent)
        (objtype ?objtype) (result nil))
(Session (roles $?roles) (id ?session))
(grant-by-role-intent (urole ?urole
                      & :(member$ ?urole ?roles))
                      (intent ?intent)
                      (objtype ?objtype))

=>
(printout t "Access Granted" crlf)
(modify ?f (result "granted")))

(deftemplate CreateSession
"Request for creating a session
id:      The desired session id
uid:     The identity of the user
roles:   The set of requested roles
groles:  Internal use, the list of roles granted so far
result:  The result of the rule evaluation"
(slot id)
(slot uid)
(multislot roles)
(multislot groles)
(slot result))

(defquery user_assigned
(declare (variables ?uid ?role))
(UA (uid ?uid) (role ?role)))

(defrule default_deny_Session
"This rule will fire if there are no other rules,
providing the default-deny behaviour"
; Don't run unless there are no other rules:
(declare (salience -100))
; fire only for rules without a result:
?f <- (CreateSession (result nil))
=>
(printout t "I'm sorry, Dave, I can't do that" crlf)
(modify ?f (result "denied")))

```

```

(deftemplate DSD "Dynamic Separation of Duty
n:      The maximum number of active roles from this set
        one session is allowed to have
roles:  The list of roles"
(slot n)
(multislot roles))

(defrule CreateSession-InitialStep
"Create a Session - First Check

This is the initial step, unless no roles were
requested. The set of rules is checked against the
DSD table, and the first role is checked against the
User Assignment.

If both checks succeed, the first role is moved to the
g(ranted-)roles list. This is repeated until all rules
have been checked against the User Assignment.

In order to not check the DSD on each step, this rule
has been copied, and the DSD check has been removed from
the copy while adding a check for a (non-)empty groles
list to each version."

?f <- (CreateSession (id ?id) (uid ?uid)
        (roles ?role1 $?roles)
        (groles /* empty */))

(UA (uid ?uid)
    (role ?role1))

; The DSD check is quite tricky: We create a union of
; the rules from the request and each DSD entry in turn.
; While creating this union, the intersecting roles are
; removed from the resulting list, therefore the number
; of duplicates is the difference between the combined
; length of the original lists and the length of the
; union.

; DSD restrictions specify a number of roles n. If at
; least this number of roles intersect with the
; requested set, the request shall be denied.

; Therefore we get |requested_roles| + |dsd_roles| >=
; |requested_roles (union) dsd_roles| + n
; for any non-conforming combinations.

(not (DSD (n ?n)
        (roles $?dsd_roles &
         :(>= (+ (length$ ?roles)
                 (length$ ?dsd_roles)

```

```

        1)
        (+ (length$(union$ (list ?role1)
                           ?roles
                           ?dsd_roles))
           ?n ))))
=>
; move the granted role to groles:
(modify ?f (groles ?role1)
          (roles ?roles))

(defrule CreateSession-NextStep
"Create a Session - Iterating Check

check the next role from the list of requested roles.
DSD checks have been done by the rule above."

; The DSD check has been done.
?f <- (CreateSession (id ?id) (uid ?uid)
                ; Need to check roles?
                (roles ?role1 $?roles)
                ; groles is non-empty?
                (groles ?grole1 $?groles)
                (result nil))

(UA (uid ?uid)
   (role ?role1))
=>
;move the granted role to groles:
(modify ?f (groles (list ?grole1 ?groles ?role1))
          (roles ?roles))

(defrule CreateSession-FinalStep
"Create a Session - Instantiate Session Fact

grant Session if all requested roles have been
processed, or if no role was requested."
?f <- (CreateSession (id ?id) (uid ?uid)
                (roles $?roles
                 & : (= (length$ ?roles) 0))
                (groles $?groles))
=>
(assert (Session (id ?id) (uid ?uid) (roles ?groles)))
(retract ?f))

```

## B Example Session

First, you need to unpack and install the Jess engine. Unless you want to write a Java application, it's enough to unpack Jess somewhere into your home directory. For

convenience, you may link the start script into your binary directory or to your working directory

Now you can load Jess by typing “jess”, preferably after changing into the directory containing the Jess source files. Jess will start and present the Jess prompt, “Jess> \_”. Assuming you have saved the above source code as rbac.jess, you can load it by typing “(batch rbac.jess)”. In order to leave Jess, you can either type “(exit)” or press ^D, the end-of-file character.

Since this rbac console does not run concurrently to the role engine, we cannot start the engine with the rbac rule set, instead we will run the engine each time we add a fact representing an event.

First, we want to load a policy. This is necessary because the default policy is to deny everything. We will create two users, “root” and “Dave Null”, both having the assigned roles “user” and “programmer”. “root” will have the additional role “admin”.

Then we create a Dynamic Separation of Duty constraint, disallowing to have programmer and admin privileges within the same session.

Finally, we grant access to any system object to the administrator, access to userhome to Dave, and read access to system objects (e.g. /etc/resolv.conf) to Dave.

```
(deffacts access-by-role-info
  (UA (uid "root") (role "programmer"))
  (UA (uid "root") (role "user"))
  (UA (uid "root") (role "admin"))
  (UA (uid "Dave Null") (role "programmer"))
  (UA (uid "Dave Null") (role "user"))
  (DSD (n 2) (roles "programmer" "admin"))
  (grant-by-role (urole "admin")
                 (objtype "system"))
  (grant-by-role (urole "user")
                 (objtype "userhome"))
  (grant-by-role-intent
   (urole "user")(intent "read")
   (objtype "system"))
)
```

If you save this policy to the file policy.jess, you can read it using (batch policy.jess). Now we want to see the defined rules facts using the (rules) and (facts) command:

```
./Jess
Jess, the Rule Engine for the Java Platform
Copyright (C) 2006 Sandia Corporation
Jess Version 7.0p1 12/21/2006
Jess> (batch rbac.jess)
TRUE
Jess> (batch policy.jess)
TRUE
Jess> (rules)
MAIN::CreateSession-FinalStep
```

```

MAIN::CreateSession-InitialStep
MAIN::CreateSession-NextStep
MAIN::access-by-role-intent-r
MAIN::access-by-role-r
MAIN::default_deny_Access
MAIN::default_deny_Session
MAIN::user_assigned
For a total of 8 rules in module
MAIN.
Jess>
(facts)
For a total of 0 facts in module MAIN.

```

As expected, all rules are printed, but there are no defined facts. The reason is: Facts defined using deffacts are only asserted if the engine is reset, while facts defined using assert are erased on engine reset:

```

Jess> (reset)
TRUE
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::grant-by-role
    (urole "admin") (objtype "system"))
f-2 (MAIN::grant-by-role
    (urole "user") (objtype "userhome"))
f-3 (MAIN::UA (uid "root") (role "programmer"))
f-4 (MAIN::UA (uid "root") (role "user"))
f-5 (MAIN::UA (uid "root") (role "admin"))
f-6 (MAIN::UA
    (uid "Dave Null") (role "programmer"))
f-7 (MAIN::UA (uid "Dave Null") (role "user"))
f-8 (MAIN::DSD
    (n 2) (roles "programmer" "admin"))
f-9 (MAIN::grant-by-role-intent
    (urole "user") (intent "read")
    (objtype "system"))
For a total of 10 facts in module MAIN.

```

Let's start a session for Dave and run the engine:

```

Jess> (assert (CreateSession (id 1)
    (uid "Dave Null") (roles "admin")))
<Fact-10>
Jess> (run)
I'm sorry, Dave, I can't do that
1
Jess> (facts)
[...]
f-10 (MAIN::CreateSession

```

```
(id 1) (uid "Dave Null") (roles "admin")
(groles ) (result "denied"))
```

Jess did not find a rule which would allow Dave to be an admin. He may choose to log in as user and programmer instead. We will inspect the actions of the rule engine using the watch command:

```
Jess> (reset)
TRUE
Jess> (watch all)
TRUE
Jess> (assert (CreateSession
              (id 1) (uid "Dave Null")
              (roles "user" "programmer")))
==> f-10 (MAIN::CreateSession
         (id 1) (uid "Dave Null")
         (roles "user" "programmer")
         (groles ) (result nil))
==> Activation: MAIN::default_deny_Session :
      f-10
==> Activation: MAIN::CreateSession-InitialStep :
      f-10, f-7,
<Fact-10>
Jess> (run)
FIRE 1 MAIN::CreateSession-InitialStep f-10, f-7,
<== Activation: MAIN::default_deny_Session :
      f-10
   <=> f-10 (MAIN::CreateSession (id 1)
           (uid "Dave Null") (roles "programmer")
           (groles "user") (result nil))
==> Activation: MAIN::default_deny_Session :
      f-10
==> Activation: MAIN::CreateSession-NextStep :
      f-10, f-6
FIRE 2 MAIN::CreateSession-NextStep f-10, f-6
<== Activation: MAIN::default_deny_Session :
      f-10
   <=> f-10 (MAIN::CreateSession
           (id 1) (uid "Dave Null")(roles )
           (groles "user" "programmer")
           (result nil))
==> Activation: MAIN::default_deny_Session :
      f-10
==> Activation: MAIN::CreateSession-FinalStep :
      f-10
FIRE 3 MAIN::CreateSession-FinalStep f-10
==> f-11 (MAIN::Session
         (id 1) (uid "Dave Null")
         (roles "user" "programmer"))
```

```

<== f-10 (MAIN::CreateSession
          (id 1) (uid "Dave Null") (roles )
          (groles "user" "programmer")
          (result nil))
<== Activation: MAIN::default_deny_Session :
      f-10
<== Focus MAIN 3
Jess> (unwatch all)
TRUE

```

When the CreateSession fact is asserted, Jess will activate each rule matching this fact. One rule is the default deny rule (using only fact f-10, this new fact), since it will match any CreateSession fact. The other rule is the CreateSession-InitialStep rule. It uses f-10 and the User Assignment f-7 from above, matching the first role. It does not yet use f-6. “<Fact-10>” is the result of the assert, like in the first example.

After starting the engine, the InitialStep rule fires, because it has the higher salience value of the two active rules. This rule modifies (“<=>”) the CreateSession fact by moving the first requested role to the list of granted roles. The default deny rule is deactivated (“<== Activation”) before modifying the CreateSession fact, and activated again after the modification is done. Additionally, the NextStep rule is activated (instead of the InitialStep rule), since the granted roles list is non-empty. The NextStep rule will use the f-6 User Assignment fact, since the new first role matches.

Again, the rule with the highest salience value, CreateSession-NextStep, is fired, and like the initial step, it moves the first role to the list of granted roles. This activates the FinalStep rule, since there are no more requested roles. This rule creates the Session fact, f11, from the given values and removes the CreateSession fact.

Now we will try to start a session for root, selecting all of his possible roles:

```

Jess> (reset)
TRUE
Jess> (watch all)
TRUE
Jess> (assert (CreateSession (id 42) (uid "root")
                          (roles "programmer" "user" "admin")))
==> f-10 (MAIN::CreateSession (id 42)
          (uid "root")
          (roles "programmer" "user" "admin")
          (groles ) (result nil))
==> Activation: MAIN::default_deny_Session :
      f-10
<Fact-10>

```

As you can see, only the deny rule is active, this session will be refused. The InitialStep rule does not match because of the DSD policy, the NextStep rule requires a non-empty groles slot and the FinalStep will wait for roles to become empty.

Now we will test if the rule set works correctly:

```

Jess> (assert (CreateSession (id 43) (uid "root")
  (roles "programmer" "user")))
==> f-11 (MAIN::CreateSession (id 43)
  (uid "root")
  (roles "programmer" "user")
  (groles ) (result nil))
==> Activation: MAIN::default_deny_Session :
  f-11
==> Activation: MAIN::CreateSession-InitialStep :
  f-11, f-3,
<Fact-11>
Jess> (assert (CreateSession (id 44) (uid "root")
  (roles "user" "admin")))
==> f-12 (MAIN::CreateSession (id 44)
  (uid "root") (roles "user" "admin")
  (groles ) (result nil))
==> Activation: MAIN::default_deny_Session :
  f-12
==> Activation:
  MAIN::CreateSession-InitialStep :
  f-12, f-4,
<Fact-12>
Jess> (assert (CreateSession (id 45) (uid "root")
  (roles "programmer" "admin")))
==> f-13 (MAIN::CreateSession (id 45)
  (uid "root")
  (roles "programmer" "admin")
  (groles ) (result nil))
==> Activation: MAIN::default_deny_Session :
  f-13
<Fact-13>

```

As expected, the FirstStep rule would only fire if the DSD permits the selected combination of roles.

One final test: Will unprivileged session be allowed?

```

Jess> (assert (CreateSession (id 46) (uid "root")
  (roles)))
==> f-14 (MAIN::CreateSession (id 46)
  (uid "root") (roles ) (groles )
  (result nil))
==> Activation: MAIN::default_deny_Session :
  f-14
==> Activation: MAIN::CreateSession-FinalStep :
  f-14
<Fact-14>

```

They will.

Now we will access some files - or be prohibited from accessing them. For this test, we will need need some sessions:

```
Jess> (unwatch all)
TRUE
Jess> (reset)
TRUE
Jess> (assert (CreateSession (id 1)
                (uid "Dave Null") (roles "user")))
<Fact-10>
Jess> (assert (CreateSession (id 2) (uid "root")
                (roles "admin")))
<Fact-11>
Jess> (run)
4
Jess> (facts)
...
f-9   (MAIN::grant-by-role-intent (urole "user")
      (intent "read") (objtype "system"))
f-12  (MAIN::Session (id 2) (uid "root")
      (roles "admin"))
f-13  (MAIN::Session (id 1) (uid "Dave Null")
      (roles "user"))
For a total of 12 facts in module MAIN.
```

Then we will assert the Access facts and run the engine:

```
Jess> (assert (Access (session 1)
                    (objtype "undefined")
                    (intent "malicious")))
<Fact-14>
Jess> (assert (Access (session 1)
                    (objtype "system")
                    (intent "read")))
<Fact-15>
Jess> (assert (Access (session 1)
                    (objtype "system")
                    (intent "write")))
<Fact-16>
Jess> (assert (Access (session 2)
                    (objtype "system")
                    (intent "write")))
<Fact-17>
Jess> (assert (Access (session 1)
                    (objtype "userhome")
                    (intent "write")))
<Fact-18>
Jess> (assert (Access (session 2)
                    (objtype "userhome"))
```

```

        (intent "write")))
<Fact-19>
Jess> (run)
Access Granted
Access Granted
Access Granted
Permission denied
Permission denied
Permission denied
6
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::grant-by-role (urole "admin")
    (objtype "system"))
f-2 (MAIN::grant-by-role (urole "user")
    (objtype "userhome"))
f-3 (MAIN::UA (uid "root") (role "programmer"))
f-4 (MAIN::UA (uid "root") (role "user"))
f-5 (MAIN::UA (uid "root") (role "admin"))
f-6 (MAIN::UA (uid "Dave Null")
    (role "programmer"))
f-7 (MAIN::UA (uid "Dave Null") (role "user"))
f-8 (MAIN::DSD (n 2)
    (roles "programmer" "admin"))
f-9 (MAIN::grant-by-role-intent
    (urole "user") (intent "read")
    (objtype "system"))
f-12 (MAIN::Session (id 2) (uid "root")
    (roles "admin"))
f-13 (MAIN::Session (id 1) (uid "Dave Null")
    (roles "user"))
f-14 (MAIN::Access (session 1)
    (objtype "undefined")
    (intent "malicious")
    (result "denied"))
f-15 (MAIN::Access (session 1)
    (objtype "system") (intent "read")
    (result "granted"))
f-16 (MAIN::Access (session 1)
    (objtype "system") (intent "write")
    (result "denied"))
f-17 (MAIN::Access (session 2)
    (objtype "system") (intent "write")
    (result "granted"))
f-18 (MAIN::Access (session 1)
    (objtype "userhome") (intent "write")
    (result "granted"))
f-19 (MAIN::Access (session 2)
    (objtype "userhome") (intent "write")
    (result "denied"))

```

For a total of 18 facts in module MAIN.

As you can see, the “Access Granted” and “Permission denied” messages do not reflect the order of the facts. Instead, all “Granted” messages are printed first, then all “denied” messages are printed. This is due to the salience value of the deny rule. Only looking on the facts reveals the outcome of each individual evaluation.

Finally, let us be curious about why one particular access is granted:

```
Jess> (reset)
TRUE
Jess> (assert (CreateSession (id 1)
                        (uid "root") (roles "admin")))
<Fact-10>
Jess> (run)
2
Jess> (watch all)
TRUE Jess> (assert (Access (session 1)
                        (objtype "system")
                        (intent "write")))
==> f-12 (MAIN::Access (session 1)
            (objtype "system") (intent "write")
            (result nil))
==> Activation: MAIN::default_deny_Access :
        f-12
==> Activation: MAIN::access-by-role-r :
        f-12, f-11, f-1
<Fact-12>
Jess> (facts)
f-0 (MAIN::initial-fact)
f-1 (MAIN::grant-by-role (urole "admin")
            (objtype "system"))
...
f-11 (MAIN::Session (id 1) (uid "root")
            (roles "admin"))
f-12 (MAIN::Access (session 1)
            (objtype "system") (intent "write")
            (result nil))
For a total of 12 facts in module MAIN.
```

As you can see, the access-by-role will use the fact representing the session and the grant-by-role fact representing the Permission Assignment; one (the) session. assigned to Session 1 and the Object Type from f-12 match f-1.

Now we have seen everything we need to know, except how to quit the session:

```
Jess> (exit)
You have new mail
$
```

## References

- [Dro] *Drools Documentation*.  
<http://www.jbug.jp/trans/jboss-rules3.0.2/ja/html/bk01-toc.html>.
- [Jesa] *The Jess Rule Engine*. <http://www.jessrules.com/jess>.
- [Jesb] *The Jess Documentation*. <http://herzberg.ca.sandia.gov/jess/docs/70>, or included in the Jess program archive.
- [RBA04] *The RBAC American National Standard*. ANSI INCITS 359-2004, 2004.