

[0059] Referring to FIG. 3, the capabilities of alice in RBAC, $caps_{RBAC}(alice) = \{(r, mrec1), (w, mrec1), (r, mrec2), (w, mrec2), (r, mrec3), (w, mrec3)\}$ because $alice \rightarrow^+ Doctor \rightarrow \{w\} \rightarrow Med\ Records$, $alice \rightarrow^+ Intern \rightarrow \{r\} \rightarrow Med\ Records$, $Doctor \rightarrow^+ RBAC$, $Intern \rightarrow^+ RBAC$ and $mrec1, mrec2, mrec3$ are assigned to $Med\ Records$ and $Med\ Records \rightarrow^+ RBAC$.

[0060] Referring to FIG. 5, a PM permission in the system 20 is a triple (u, op, o) where u is a user, op is an operation, and o is an object, and for each policy class pc_k under which o is protected, indicating that the user u has an attribute ua_k in pc_k , object o has an attribute oa_k in pc_k , and there exists an operation set ops_k containing op that is assigned to both ua_k and oa_k .

[0061] With respect to FIG. 3, the triple $(alice, w, mrec2)$ is a permission, because (1) $mrec2$ is contained in both RBAC and MLS, (2) both Doctor and Secret are alice's user attributes, where Doctor is contained in RBAC and $mrec2$ is contained in MLS, (3) S_TS and Med Records are object attributes of $mrec2$, where Med Records is contained in RBAC and S_TS is contained in MLS, and (4) the operation set $\{w\}$ is assigned to both Doctor and Med Records and is assigned to both Secret and S_TS. In contrast, the triple $(bob, w, mrec2)$ is not a permission, because $mrec2$ is contained in both RBAC and MLS, but bob does not have an attribute in MLS.

[0062] Administrative Operations

[0063] An administrative operation simply creates, deletes, or modifies an existing policy state data relation. The set of administrative operations include, for example, create/delete user, create/remove assignment, etc. The state of the overall PM policy changes as a consequence of the execution of an administrative operation. The administrative operations are executed on behalf of a user via administrative commands (users never execute operations directly), or automatically by the system 20 in response to a recognized event. Event-response relations are described in the following section.

[0064] An administrative operation could be specified as a parameterized procedure, whose body describes how a data set or relation (denoted by R) changes to R':

```

opname(x1, . . . ,xk) {
  R' = f(R, x1, . . . ,xk)
}

```

[0065] For example, consider the following administrative operation CreateUser:

```

CreateUser(u) {
  U' = U ∪ {u}
}

```

[0066] The CreateUser administrative operation specifies that the creation of a new user with the identifier "u" consists of augmenting the user set U with the new user identifier. Included in this specification is the fact that if a user with the same identifier already exists, the operation has no effect.

[0067] An administrative command is a parameterized sequence of administrative operations prefixed by a condition and has the format:

```

commandName(x1, . . . ,xk)
if (condition) then
  paop1
  . . .
  paopn
end

```

[0068] where x_1, \dots, x_k ($k \geq 0$) are (formal) parameters and $paop_1, \dots, paop_n$ ($n \geq 0$) are primitive administrative operations which may use x_1, \dots, x_k as their parameters. The condition tests, in general, whether the user who requested the execution of the command is authorized to execute the command (i.e., the composing primitive operations), as well as the validity of the actual parameters. If the condition evaluates to false, then the command fails. For example, the command that grants a user attribute a set of operations on an object container could be defined as follows:

```

grant(cert_process, ua, oa, op1, . . . ,opm)
if (uaeUA ∧ oaεOA ∧
  ∀iε1..m op_iεOP ∧
  ops ⊆ OP ∧
  is_auth(cert_process.user, create_opset) ∧
  is_auth(cert_process.user, oattr_assign_opset_to, oa) ∧
  is_auth(cert_process.user, uattr_assign_to_opset, ua)) then
  create_opset(ops, op1, . . . ,opm)
  assign_opset_to_attr(ops, oa)
  assign_attr_to_opset(ua, ops)
end

```

[0069] For convenience, a command may exist inside another command.

[0070] Prohibitions

[0071] Permission relations alone are not sufficient in specifying and enforcing the current access state for many types of policies. Other policies pertain to prohibitions or exceptions to permissions. Deny relations specify such prohibitions. System 20 deny-relations take on two forms, user-based deny and process-based deny. User-based deny relations associate users with capabilities (op, o) that the user and the user's subjects are prohibited from executing. For example, although a user with the attribute IRS Auditor may be allowed to review IRS tax records, a user-based deny relation could prevent that user from reviewing his/her own tax record. Process-based deny relations associate processes with capabilities (op, o) that the processes are prohibited from executing. Process-based deny relations are usually created through the use of obligations (see below). User-based deny relations can be created either through administrative commands or through obligations.

[0072] A user-based deny relation is a triple $\langle u, ops, os \rangle$, where $u \in U$, $ops \in 2^{OP}$, and $os \in 2^O$. The meaning of the user-based deny is that a process executing on behalf of user u cannot perform any of the operations in ops on any of the objects in os . The set of user-based deny relations is denoted as UDENY in FIG. 2.

[0073] A process-based deny relation is a triple $\langle p, ops, os \rangle$, where $p \in P$, $ops \in 2^{OP}$, and $os \in 2^O$. The meaning of the process-based deny is that the process p may not perform any of the operations in ops on any of the objects in os . The set of process-based deny relations is denoted as PDENY in FIG. 2.