

respect to an MLS policy, the elements of the dominance relation could be parameterized, and the specific levels could be defined just prior to importation, or with respect to a DAC policy, delegation details could be defined.

[0154] The following is an example of how system **20** can be configured to offer support to an application. One type of application provides services that are independent of access control. These applications include, for example, text editors, spreadsheets, and drawing packages. Another type of application provides services through the use of an access control policy. For example, e-mail applications provide for the reading of messages and attachments through the discretionary distribution of objects, and workflow management applications provide for the reading and writing of specific documents by a prescribed sequence of users.

[0155] In the following example, the system **20** is configured to support a simple workflow application. In this example, the following activities and users or roles perform those activities sequentially.

[0156] Activity 0. A user in the "Secretary" role fills out a purchase order form and attaches a "routing slip" that specifies $n \geq 1$ users and/or roles and the order in which they must approve and sign the purchase order.

[0157] Activity $k=1$ to n : The user or a user in the role specified in the routing slip at position k approves and signs the purchase order.

[0158] Activity $n+1$: A user in the "Acquisition" role examines the purchase order before ordering the items.

[0159] It is assumed that the workflow policy also imposes the restriction "No user is allowed to sign a purchase order twice".

[0160] In the following, we describe the system **20** configuration used to specify and enforce the policy described above.

[0161] First, the purchase order will be modeled by an object. Activity 0 performed by a user in the Secretary role consists of reading an empty form object, filling out the form and creating a purchase order object with the data from the form. Each signing activity consists of reading the purchase order object from the specified user's or role's work items, applying maybe a graphic and/or electronic signature to its content, and writing back the purchase order object. Finally, activity $n+1$ consists of simply reading the purchase order object. The purchase order object will not be accessible to a user unless all previous activities as specified in the sequence have been successfully completed. Note that the policy enforcement will be performed by the OS kernel, not by the application.

[0162] The system **20** configuration will include two policy classes, DAC and RBAC. The DAC policy will comprise the user identifiers, the user name attributes, and a work items object container for each user. Each user has read/write access to its work items.

[0163] The RBAC policy will comprise the user identifiers, the Secretary role, a few "signing" roles, and the Acquisition role. The Secretary role has read access to a blank purchase order form included in the Forms container, write access to a container of Completed Forms, and the privilege of composing and registering event-response relations with the system **20**. Each signing role has read/write access to its work items. The Acquisition role has read access to the container of Approved Orders. FIGS. 10a-e illustrate a configuration with three signing roles (Accounts Receivable, Contracting, and Accounts Payable) and three signing users (alice, bob, and charlie), a Secretary user, katie, and an Acquisition user, dave.

[0164] The activity sequencing will be ensured by the system **20** changing the purchase order location after each successful completion of an activity. Behind the automatic

moves performed by the system **20** is an event-response script composed and registered with the system **20** by the workflow application running on behalf of the user acting in the Secretary role, just before the creation of a new purchase order.

[0165] Processing of a purchase order starts with the user katie in the Secretary role filling the empty form, attaching a routing slip, and saving the form in the "Completed Forms" container as object po121 for example. The same user also generates $n+1$ event-response relations that specify what should happen after the successful completion of each of the activities 0, 1, . . . , n .

[0166] This example assumes that the routing slip as composed by katie contains, in order, user alice, role Contracting, and role Accounts Payable. The event-response relation corresponding to the successful completion of activity 0 might look as follows, assuming that the first on the routing slip is user alice:

[0167] R_0 : write object po121 in "Completed Forms" \Rightarrow assign crt_object to "alice work items"; delete assignment of crt_object to "Completed Forms."

[0168] For an activity k with $k \in 1..n-1$, the corresponding event-response relation might look as follows:

[0169] R_k : write object po121 in "Role/user $_k$ work items" \Rightarrow assign(crt_object, "Role/user $_{k+1}$ work items"); delete assign(crt_object, "Role/user $_k$ work items"); create deny(crt_user, {w}, crt_obj).

[0170] The last administrative command prevents a user from signing twice the purchase order (actually, the user could sign the order but not save it back). Finally, for activity n the event-response relation may be as follows:

[0171] R_n : write object po121 in "Role/user $_n$ work items" \Rightarrow assign(crt_object, "Approved Orders"); delete assign(crt_object, "Role/user $_n$ work items"); delete event/response(R_0, \dots, R_n).

[0172] This sends the purchase order to the "Approved orders" container, from where the Acquisition role can read it. The last command in this relation also deletes all event-response relations related to this purchase order object. For our example, the event-response relations are:

[0173] R_0 : write object po121 in "Completed Forms" \Rightarrow assign crt_object to "alice work items"; delete assignment of crt_object to "Completed Forms".

[0174] R_1 : write object po121 in "alice work items" \Rightarrow assign(crt_object, "Contracting work items"); delete assign(crt_object, "alice work items"); create deny(crt user, {w}, crt_obj).

[0175] R_2 : write object po121 in "Contracting work items" \Rightarrow assign(crt_object, "Accounts Payable work items"); delete assign(crt_object, "Contracting items"); create deny(crt user, {w}, crt_obj).

[0176] R_3 : write object po121 in "Accounts Payable items" \Rightarrow assign(crt_object, "Approved Orders"); delete assign(crt_object, "Accounts Payable work items"); delete event/response(R_0, R_1, R_2, R_3).

[0177] As noted before, when alice performs activity 1, right after she saves the signed purchase order in her work items container, the system **20** generates a deny (alice, {w}, po121), according to the event-response relation R_1 . If alice tries to sign the purchase order again as a member of the Accounts Payable role in Activity 3, she would be prevented from saving the purchase order by the above deny. Only Charlie would be able to sign po121 for the Accounts Payable role.

[0178] The system **20** provides benefits over the existing access control paradigm. For instance, the system **20** provides policy flexibility. Virtually any collection of attribute-based access control policies can be configured and enforced (e.g.,