ADMINISTRATION RIGHTS IN THE SDSD-SYSTEM

Joachim Biskup
University of Dortmund
44221 Dortmund
Germany
biskup@ls6.cs.uni-dortmund.de

Thomas Leineweber
University of Dortmund
Germany
Thomas.Leineweber@uni-dortmund.de

Jörg Parthe
University of Dortmund
Germany
parthe@ls6.cs.uni-dortmund.de

Abstract The SDSD-system offers state-dependent access control in distributed object systems. The system enforces protocols which declare sets of activity sequences as allowed, thereby forbidding any occurence of an activity outside the context of an allowed sequence. In this paper we conceptually extend the SDSD-system by introducing discretionary administration rights for controlling the activities of declaring, binding and starting SDSD-protocols. Additionally, we introduce second level administration rights for controlling the grant and revoke activities concerning administration rights. Exploiting the powerful potentials of the SDSD-system, the new concepts are implemented by special administration protocols. Administration protocols are enforced with the same techniques already used for application protocols concerning the functionality of the underlying object system. In this environment recursive revocation is shown to be feasible.

Keywords: Security, distributed object system, administration of access rights, access control, state-dependent access rights, revoking rights, recursive revocation

1. Introduction

Access control is a fundamental mechanism for enforcing security requirements in computing systems. Conceptually, and highly simplified, *access control* comprises two phases. In the first phase, one or several security administrators are *declaring access rights,* each of which state that some user is allowed to perform some action on some object. In the second phase, each user request is intercepted by a *monitor* that decides on the basis of the declared access rights whether or not the request actually shall be executed. In implementations, usually these phases are interwoven, and participants of the computing system can act both as administrator and as user. The basic approach of access control has been refined in various ways in order to deal with more specific aspects. In this paper we deal with the following aspects: state-dependent access rights, monitoring for distributed systems, and constraining the administration of access rights by access control in turn.

Concerning the first two aspects, we rely on our previous work [1, 5, 2] about the design of *state-dependent* access control and its enforcement in *distributed object systems* and the corresponding implementation by the *SDSD-system* (for *S*tate-*D*ependent *S*ecurity *D*ecisions). Varying the basic approach, access rights are no longer statically granted until an explicit revocation. Rather, in a conceptually first phase an administrator declares so-called *protocols* in order to specify which kinds of *activity sequences* of the distributed object system are considered to be allowed for formal participants. After instantiating a protocol by binding actual participants to the formal participants and starting a concrete instance of such an activity sequence, the distributed monitoring system assigns and withdraws *dynamic access rights* on a short term basis for just a single activity, thereby enforcing that only allowed sequences can actually occur.

For example, an administrator for an insurance company can allow activity sequences of the following form (which informally circumscribes a formal protocol presented in Figure 2 of [2]): each such sequence starts by drafting a contract, followed by a careful inspection with an acceptance decision; dependent on this decision: either the contract is confirmed and subsequently either it becomes valid based on a timely payment or it is rejected based on a missed payment deadline; or the contract is immediately rejected. While enforcing state-dependent access control, for instance the activity "contract rejection" cannot occur as an isolated event, rather it is dynamically enabled only just after the pertinent event of "missed deadline" or "negative inspection", respectively, and after an actual execution, the activity "contract rejection" is immediately blocked again.

The present paper aims at extending the SDSD-system with respect to the third aspect mentioned above, namely, how in turn to *control the administration* of allowed activity sequences. More specifically, we first address the most

urgent problem: *How to allow participants of the system to act as administrator who can declare protocols, bind actual participants to an instantiated protocol and start a concrete activity sequence?* We even go one level further and address the immediate follow-up problem: *How to allow participants to allow other participants to act as administrator?* Additionally, we also consider the revocation problem for both levels: *How to revoke allowances to act as administrator or to grant such allowances, respectively?*

In terms of our example, we deal with the problem how one or more "administrators" are dynamically introduced and controlled, and how a protocol of the sketched kind is coming into existence and exploited in a controlled fashion. Abstractly speaking, we investigate the problem of *administration rights* within the framework of state-dependent access control in distributed object systems. This problem has been studied before within different frameworks, including theoretical studies on deciding the possibility of the proliferation of a right (showing undecidability in general [8] and decidability of special cases like the take-grant model [10]), mechanisms for dynamic rights amplification in systems like UNIX (by the suid-flag) [6] or Hydra [17], a variety of revocation options as in Oracle [11, 12] or role-based administration of roles [16, 4].

Though our solutions are original for our framework, as discussed in detail in Section 6, they are based on an established paradigm, namely, to smoothly integrate the control of the administration with the control of the primary functionality. Furthermore, we follow the well-known paradigm of discretionary access control. More specifically, the contributions of this paper can be summarized as follows: (1) By re-examining our SDSD-system, we identify *fundamental administration tasks,* namely protocol *declaration,* protocol (instance) *binding,* and protocol (instance) *starting* (Section 2). (2) Based on the literature, we set up a conceptual design with discretionary *administration rights* as first level rights for the fundamental administration tasks and corresponding *second level rights* for controlling those of the first level. This design also includes the notion of *ownership* and control of *revocations* (Section 3). (3) Exploiting the powerful potentials of the SDSD-system, we implement the design by defining appropriate *administration protocols* for the SDSD-system, which are then shown to enforce the conceptual administration rights by the (slightly extended) mechanisms of the SDSD-system (Section 4). Some details of the implementation are demonstrated by considering the "GrantGrant right" as an example (Section 5). (4) We sketch that even *recursive revocation* can be integrated into the extended SDSD-system. (5) The implementation is available as a *prototype,* implemented in JAVA and CORBA.

On first sight, our contributions seem technically specific for the SDSD-system. However, we emphasize that – independently of the specific control system under consideration – the administration of rights always does not only have static implications, but also dynamic ones.

2. The SDSD-System

The SDSD-system has been presented in [2] and its theoretical foundation in [1, 5]. Here, we will give only a short overview of the system. The SDSD-system realizes a state-dependent access control in a distributed object system. The monitored objects communicate via a *CORBA Object Request Broker.* Every monitored object (also called *functional object*) is wrapped by its own *security object.* The functional object and its security object form a so-called *(actual) participant.* A method call is permitted if an appropriate dynamic right is present. These rights are automatically granted and revoked by the system dependent on the current state and based on *allowed activity sequences.* The building components of these allowed activity sequences are triples consisting of an *activator,* an *executor* and an *action.* Such a triple, named *step,* expresses that the participant which acts as the executor performs an action (i. e. a method call on his functional object) on behalf of the participant which acts as the activator.

The specification of the allowed activity sequences is done with a so-called *protocol.* A formal definition of the language for protocols is given in [3]. The core of a protocol is a regular expression on steps. A step specifies a building component of an activity sequence and accordingly comprises an identifier for the activator, an identifier for the executor and an action (step rule). The latter is specified by an action identifier (first action rule) possibly followed by appropriate parameters (second action rule and param rules). Though desirable in any protocol, parameters have not been generally supported in the prototype, but they are needed and actually implemented for the specific types of protocols introduced in Section 4. The identifiers used to determine activator and executor of a step have to be declared in the remainder of the protocol and are the so-called *formal participants.* These are placeholders for the actual participants, who act as activator and as executor, and have to be bound before a concrete activity sequence is monitored.

When a new activity sequence should be started in the context of monitoring an application, several administration activities additionally have to take place. Some of them are under the control of the users of the application, others are done by the system, transparently for the users. Figure 1 gives an overview of the main activities and the order of their execution. The gray arrows and boxes show the functionality of the basic SDSD-system. The black arrows and the white boxes show the extensions introduced in Section 3 (et. seqq.).

First, some user has to specify the allowed activity sequences by declaring an *application protocol.* Next, the protocol has to be instantiated and bound. Therefore, another or the same user as before performs the binding activity. Thereby, he assigns actual participants to the formal participants considering the type constraints listed in the application protocol. As result of this activ-

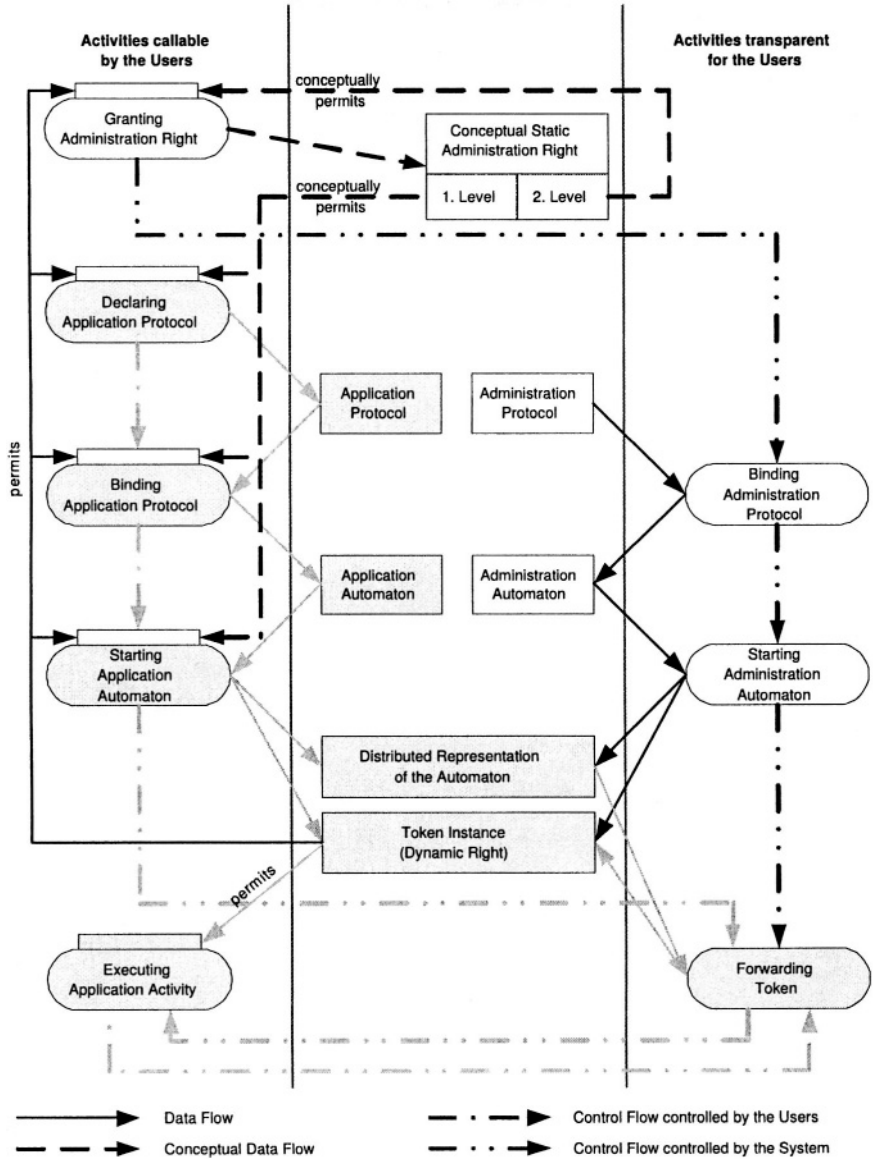

Figure 1. Overview of the main administration activities (declaring, binding, starting and granting) and the order of their execution. The functionality of the basic SDSD-system is represented by the gray arrows and the gray boxes, the proposed extensions are represented by the black arrows and the white boxes.

ity, a *finite automaton* is generated, which accepts exactly those sequences of steps (respectively the corresponding activity sequences) that are specified by the regular expression of the protocol. In the remainder this finite automaton is called *application automaton.* A declared application protocol can be instantiated and bound several times. After generating an application automaton, it has to be started. Therefore, some user performs the starting activity, which has two effects. First, a distributed representation of the application automaton is created and sent to the participants which are involved in that automaton. Second a *virtual security token* "is created".

Abstractly seeing, this token dynamically permits the execution of an application activity. Therefore, it is automatically forwarded to the participant who acts as the activator of the step which shall be executed next. The selection is taken by using the distributed representation of the automaton and is described at the end of this section. During the forwarding process the token is slightly changed, accordingly, in Figure 1 it is called token instance. After completion of an application activity the token is appropriately modified and forwarded again according to an activity sequence specified in the application automaton. In some sense, this token represents a *dynamic right,* since it is revoked immediately after single usage. After forwarding the token, the execution of a next application activity is permitted and so on. When the automaton reaches a final state the sequence is finished, thus the token and the distributed representation of the automaton are deleted.

The token and the process of forwarding it from one participant to another – as described above – are an abstract explanation of the actually used access control mechanism. Technically seeing, the security objects of the participants negotiate which application activity is permitted next by interchanging predefined messages. At first, the executor of the last executed step (called *last executor*) sends *offer* messages to the activators of possible next steps (according to the application automaton). If the functional object of an activator wants to execute the action belonging to the step, the activator answers with a *get* message. The last executor replies to one of the get messages he receives with a *put* message. This behaviour corresponds to forwarding the token in our abstract explanation. The receiving activator sends an *invoke* message to the executor of the belonging step. The invoked executor executes the action according to the step description. Thus, this step (respectively this application activity) is completed. Then the invoked executor sends – as the new last executor – new offer messages and so on.

All messages are interlinked by sequence numbers (this, among others, changes the token). Furthermore, messages as well as the application automatons are cryptographically signed and locally logged by the security objects to detect security violations.

3. Considered Problem and Approach

The SDSD-prototype, as reported in [2], only monitors the application activities but not the additional administration activities which might be crucial for achieving overall security: (1) Declaring Application Protocol, (2) Binding Application Protocol and (3) Starting Application Automaton. In Figure 1 this situation is indicated by showing a (gray) "monitoring box" on top of "Executing Application Activity" but not on top of the listed administration activities of the basic SDSD-system (the gray rounded boxes). In this paper we investigate in depth the problem of how the listed administration activities can be monitored by an extended SDSD-system, too. In terms of Figure 1, we will provide (the white) "monitoring boxes" for the administration activities such that application activities and administration activities are homogeneously supervised within the new SDSD-system.

We reconstruct the SDSD-system in a way, that the system offers so-called *administration operations,* which are used within the administration activities to accomplish the administration tasks. These operations are: (1) declare (a new application protocol), (2) undeclare (an application protocol), (3) bind (participants to an application automaton), (4) remove (a bound, but not yet started application automaton), and (5) start (an already bound application automaton).

As shown in Figure 1, the execution of an administration activity is monitored by the system (indicated by the boxes above the corresponding activity symbols; undeclaration and removal are not shown there). On a conceptual layer, these administration activities are permitted by first level static administration rights. These administration rights are granted and revoked – discretionarily – by the participants, sometimes directly with the aid of one of the two additional administration operations (1) grant (an administration right) and (2) revoke (an administration right, more exact: revoke a grant), or sometimes as a side effect of one of the other administration operations. The holder of an administration right is able to use it until the grantor revokes the originating grant as an explicit action by calling the revoke operation. Calling the grant or the revoke operation (revoke is not shown in Figure 1) is an administration activity itself, thus it is also monitored by the system. Conceptually seeing, their execution is permitted by a second level static administration right.

The set of administration right types is shown in Table 1. The *GrantGrant* right permits the execution of the administration activity grant, whereby the GrantGrant right, the GrantDeclare and the Declare right can be granted to any other participant of the system. The holder of the GrantGrant right is also allowed to revoke the grants caused by himself. The *GrantDeclare* right is a limited version of the GrantGrant right, i. e. granting a right and revoking a grant is restricted to the Declare right. A participant needs the *Declare* right to declare new application protocols or undeclare application protocols that he

Right	*Abbreviation*	*Permits*
GrantGrant	`gg`	granting gg, gd, d, resp. revoking corresponding grants caused by the holder of gg
GrantDeclare	`gd`	granting d, resp. revoking corresponding grants caused by the holder of gd
Declare	`d`	declaring new protocols resp. undeclaring protocols caused by the holder of d
Own	`o[Protocol prot]`	granting `b[prot]`, `s[prot, Automaton aut]`, `s[prot]`, resp. revoking corresponding grants
Bind	`b[Protocol prot]`	instantiating protocol `prot` (incl. binding) resp. revoking corresponding not yet started instances caused by the holder of `b[prot]`
Start	`s[Protocol prot]`	starting automatons that are instances of protocol `prot`
Start(Instance)	`s[Protocol prot, Automaton aut]`	starting automaton `aut` that is instance of protocol `prot`

Table 1. Summary of the administration right types.

has previously declared. The declarant of a protocol becomes the owner of it, i.e., he gets automatically the *Own* right. The owner of a protocol is allowed to grant all rights w.r.t. the protocol, namely, the *Bind* right and the two different types of the *Start* right and to revoke corresponding grants. The Bind right is necessary to instantiate an application protocol, i.e., to generate an application automaton and bind participants to this automaton. The first type of the Start right is solely referring to a protocol `prot`. The holder of this right is allowed to start any automaton that is an instance of `prot`. The other type of the Start right is additionally referring to a specific automaton instance `aut`, i.e., only this automaton instance can be started. Both types do not permit the abortion of any automaton, no matter who has started it. In contrast, the holder of the Bind right is allowed to remove automatons that he has previously bound. But if such an automaton is already started, its removal is forbidden.

4. Implementation

In Section 3 we have introduced conceptual static administration rights, which conceptually permit the execution of administration activities. These conceptual rights are different in kind from the dynamic rights (token) described in Section 2. The administration rights are explicitly granted and revoked by executing an appropriate administration activity. Furthermore, they can be repeatedly used (considering resource restrictions). In contrast, the dynamic rights are automatically assigned and withdrawn by the system. The withdrawal occurs directly after single usage.

To overcome these differences, we introduce a set of specialized protocols, called *administration protocols.* As shown in the right part of Figure 1, when a participant grants an administration right to another participant (or to himself), an appropriate administration protocol is chosen and bound (both within the binding administration protocol activity). The *administration automaton* generated in that process has two purposes: (1) It permits the grantee (holder) to execute all administration activities that are allowed by the granted administration right (e.g. grant an administration right to a participant on his own). (2) It permits the grantor to execute the activity "revoke the granted right, represented by this automaton". (Actually, as shown in Section 5, a special participant is allowed to end the automaton on behalf of the grantor.)

Next, this administration automaton is started. Analogously to starting an application automaton, a distributed representation of the automaton and a token (instance) are created. From this stage forth, the access control of administration activities is enforced almost in the same manner as the control of application activities. Only the selection which activator gets the put message is taken differently, namely on the basis of assigned priorities. As shown in Figure 1, binding an administration protocol and starting an administration automaton is not monitored by the system, since these activities are only initiated and executed by the system itself.

Apart from the participants mentioned in Section 2 (in the following called *functional participants*), other so-called *special participants* participate in administration automatons. These participants provide the operations that are used to carry out the administration tasks. There are three different types of special participants: *ProtocolManager, ProtocolStarter* and *GrantManager.* Participants of the first type, for simplification also called ProtocolManager (ProtocolStarter and GrantManager, respectively), offer the operations for declaring (`declare`) and undeclaring (`undeclare`) application protocols. Operations for instantiating application protocols (`bind`) as well as starting (`start`) and removing (`remove`) application automatons are provided by the ProtocolStarters. GrantManagers offer the operations for granting (`grant`) administration rights and revoking (`revoke`) corresponding grants. Furthermore, they gather the information on behalf of the functional participants, which administration rights those got, by whom and at what time. Each functional participant has exactly one GrantManager assigned to him. But one GrantManager can be assigned to several functional participants. Special participants do not have a functional object but only the (normally wrapping) security object. They do not need any user interaction. Thus, they can be settled on any host, particularly on especially secured hosts. Due to this property, they can be protected against malicious changing, therefore they are trustworthy participants. This trustworthiness is necessary, because crucial security checks have to be done in the context of the administration activities.

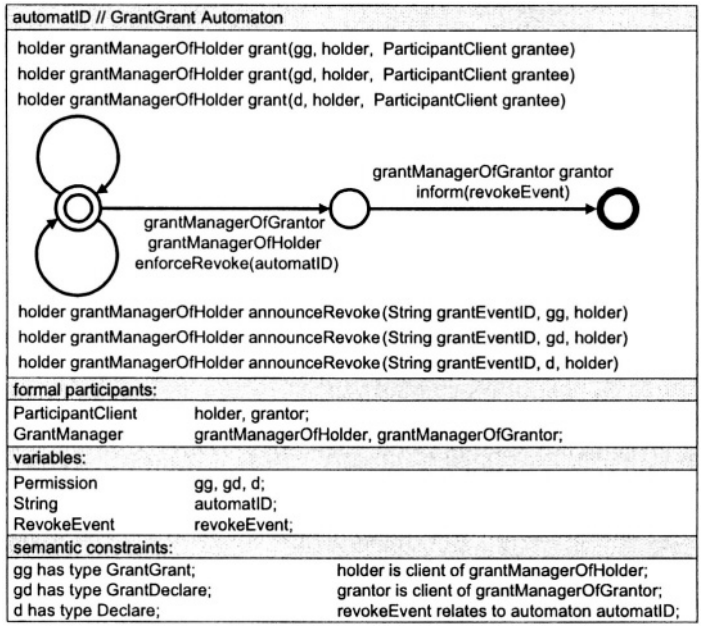

Figure 2. Automaton for the GrantGrant administration protocol

5. Some Details for Granting the GrantGrant Right

Granting a right is done by invoking the grant operation on a GrantManager. During the execution of the grant operation an appropriate administration protocol is bound to get an administration automaton. This administration automaton is then started. In this section we show some details of the GrantGrant right. The other rights introduced in Section 3 are treated similarly, as described in [13].

The administration protocol for the GrantGrant right is shown as an automaton in Figure 2. At this stage, the description of the protocol contains some additional information, which will be used during the process of creating the administration automaton and binding participants to the automaton. This information consists of (1) a list of the formal participants, (2) a list of variables and (3) a set of semantic constraints. The list of the formal participants specifies the types of the participants, who have to be bound to the automaton. In the case of the GrantGrant automaton these are (i) the `grantor`, (ii) the GrantManager that is assigned to the `grantor` (`grantManagerOfGrantor`), (iii) the grantee called `holder` and (iv) his GrantManager (`grantManagerOfHolder`). The "assigned to" relations are guaranteed by testing the two `is client of` semantic constraints. In contrast to general practice, separation of duties is not

enforced in the case of binding an administration automaton. This is necessary in the case of granting a right to oneself, when the grantor and the grantee are identical. The list of variables defines variables to which values (resp. objects) have to be assigned during the binding process. Inter alia, these are three objects `gg`, `gd` and `d`, one for every right that can be granted by holding the GrantGrant right. The `has type` semantic constraints guarantee that every object has the correct type according to its name as specified in Table 1. The variable `automatID` records the unique identifier for this automaton. The variable `revokeEvent` is used during the revocation of a right.

After binding, the GrantGrant automaton is started, i. e. execution of a first step is offered to the eligible participants. The `holder` receives six offer messages (see Section 2 for an explanation) to execute a step, one for each step in which the `holder` can act as the activator. Three offer messages apply to the steps with the parameterized action `grant`, and three to the steps with the parameterized action `announceRevoke`. The `grantManagerOfGrantor` gets one offer for the step with the parameterized action `enforceRevoke`. He accepts this offer only if his client, i.e. the `grantor`, has instructed him to revoke the grant that has led to this automaton. If he accepts, the `enforceRevoke` step has the highest priority so that it will be the next step in this automaton.

By choosing one of the three offered `grant` steps the `holder` is able to grant the GrantGrant, the GrantDeclare or the Declare right to any functional participant. The `grant` operation offered by the GrantManager of the holder expects three arguments. In each case the first two are already set to values that are determined during the binding process. Considering the preceding type information (`ParticipantClient` is the common supertype of all functional participants) the argument for the last parameter `grantee` can be freely chosen.

By predetermining some values during the binding process, the `holder` is restricted to choose the parameters for the grant operation suitably according to the description of the right he holds. In the present case the first parameter states the right that could be granted. With the aid of the second one, the invoked GrantManager is informed about the fact that the passed `holder` will be the grantor of the grant activity that should be performed under the GrantGrant right held by the `holder`. As result of granting the right, the invoked GrantManager generates and starts a new administration automaton. In that second automaton the grantee of the grant acts as the `holder` and the grantor of the grant (the `holder` in the first automaton) as the `grantor`.

If in the first automaton the `holder` wants to revoke a grant that he has previously caused, he invokes one of the three `announceRevoke` steps, depending on the granted right. The only argument that has to be handed over by the holder is a value for the parameter `grantEventID`. The invoked GrantManager of the holder expects an identifier of an automaton that was caused by the grant of a right. The `holder` gets such identifiers as a return value of the `grant` opera-

tion. The other two already set arguments ensure that the `holder` is just able to revoke automatons that are caused by his grants.

The name of the operation `announceRevoke` expresses the fact that at this point the GrantManager (of the `holder` in the first automaton) is merely charged with the revocation. The actual enforcement is done in another automaton, because revoking a grant means to end the automaton that resulted from this grant. Therefore the GrantManager keeps the necessary information in his internal attributes and waits for an offer to execute the `enforceRevoke` step in the automaton, that should be ended. Normally, he has already got such an offer, since this step, resp. its edge, begins at the initial state of the automaton. If the `holder` of the second automaton (who is the `grantee` in the `grant` step of the first automaton) is accepting his offer, i.e. he executes a `grant` or an `announceRevoke` step, the offer for the `enforceRevoke` step has been invalidated. But after completion of the `grant` or `announceRevoke` activity, the automaton will be transferred to the initial state and inter alia an offer for the `enforceRevoke` step will be sent again. When the `enforceRevoke` operation is executed, the GrantManager gets to know that the `holder` of the second automaton (who is assigned to the GrantManager) will lose a right soon. Thus, the GrantManager examines within this operation, whether further revocations have to be recursively done. Further details of the revocation are presented in [3], which also contains a concrete example for using the GrantGrant right.

6. Related and Further Work

The idea to administrate an access control system like the SDSD-system with the help of itself is very appealing, but there are not many systems with this property. Mostly, a further higher level access control system is used to administrate a basic access control system. In [7] the access control system is explicitly decomposed into an enforcement manager and a policy manager. In contrast, in the SDSD-system, the administration of the policy is done with the help of the system itself.

A widely known self-administrable access control system is Role-Based Access Control (RBAC, [14]). The administrative layer is ARBAC97 [15] respectively ARBAC99 [16] (Administrative Role-Based Access Control). It is based on and used for a decentralized administration of Role-Based Access Control systems. The administration system is partitioned into different components. In ARBAC these components are for user-role assignments, permission-role assignments and role-role assignments. These components consist of administration concepts to create, modify and remove those assignments. In the SDSD-system, there are similar assignments, but as a difference, the activities leading to these assignments are controlled. Another model for self-administration of RBAC is SARBAC (Scoped Administration of Role-Based Access Control,

[4]). It is also uses RBAC for the control of a RBAC-system, but is somewhat different to handle.

An example for a system with dynamic internal administration of rights is Hydra [17]. It assigns rights to runtime objects (of the Hydra type `local namespace`) based on rights of the originating static object and based on rights of the calling object. The UNIX operation system has also the concept of dynamic rights amplification by the suid- and the guid-flag [6]. It allows the owner of a program to give other users additional rights when executing this program. These additional rights are limited to the rights the owner of the program already has. But if the owner of the program wants that a second user shall grant rights on the program to a third user, the ownership of the program has to be transferred to the second user.

The separation into the different administration activities Declare, Bind and Start has its similarities in Workflow Management Systems [9]. There, the administration tasks involved in the creation of a new workflow are the specification of a workflow process definition itself and the creation and the start of a workflow process instance. The declaration of a SDSD-protocol corresponds to the specification of a new workflow process definition, and the process of binding and starting corresponds to the creation and start of a workflow process instance. In contrast to the Workflow Reference Model, our model explicitly distinguishes between the two tasks of creating and starting a new process instance.

Some further work has also to be done: It may be feasible to provide the system with the ability to transfer the ownership of declared protocols to another participant or to separate the ownership of a protocol from the right to grant the Bind or the Start right by introducing some additional administration rights.

Our current prototype supports only one ProtocolStarter, one ProtocolManager, one place to store the declared application protocols (Protocol Repository) and one place of the not yet started application automatons (Protocol Instance Repository). A policy for the visibility aspects has to be specified and an appropriate mechanism has to be developed to overcome this limitation. Furthermore, it is desirable that the system supports the abortion of an already started automaton. As a first idea, the protocol definition language can be expanded to mark states of the automaton (maybe called checkpoints) at which it can be aborted in a safe manner. Another important administration activity which is not yet included in our system is the introduction of new participants into the SDSD-system. This activity also might be controlled by another administration right.

References

[1] Joachim Biskup and Christian Eckert. About the enforcement of state dependent security specifications. In T. F. Keefe and C. E. Landwehr, editors, *Database Security VII,* pages

3–17. North-Holland, 1994.

[2] Joachim Biskup and Thomas Leineweber. State-dependent security decisions for distributed object-systems. In M. S. Olivier and D. L. Spooner, editors, *Database and Application Security XV,* pages 105–118. Kluwer, 2002.

[3] Joachim Biskup, Thomas Leineweber, and Jörg Parthe. Administration Rights in the SDSD-System. Technical report, University of Dortmund, 2003. Link at `ls6-www.cs.uni-dortmund.de/issi/publications/2003.html`.

[4] Jason Crampton. *Authorization and Antichains.* PhD thesis, University of London, February 2002.

[5] Christian Eckert. *Zustandsabhï¿½gige Sicherheitsspezifikationen und ihre Durchsetzung.* Berichte aus der Informatik. Shaker Verlag, 1997.

[6] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security.* O'Reilly, 1996.

[7] Robert Grimm and Brian N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Trans. on Computer Systems,* 19(1):36–70, 2001.

[8] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM,* 19(8):461–471, 1976.

[9] D. Hollingsworth. The workflow reference model. Document No. TC00-1003, 1995. Issue 1.1.

[10] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM,* 24(3):455–464, 1977.

[11] Oracle Corp. *Oracle9i Database Administrator's Guide,* 2002. Part No. A96521-01.

[12] Oracle Corp. *Oracle9i SQL Reference,* 2002. Part No. A96540-01.

[13] Jörg Parthe. Deklaration und Ausführung neuer Handlungsabfolgen fr zustandsabhängige Handlungsrechte. Diploma thesis, Universität Dortmund. `ls6-www.cs.uni-dortmund.de/~parthe/downloads/par02.pdf`, 2002.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer,* 29(2):38–47, February 1996.

[15] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. on Information and System Security,* 2(1):105–135, 1999.

[16] Ravi Sandhu and Qamar Munawer. The ARBAC99 model for administration of roles. In *15th Annual Computer Security Applications Conference,* pages 229–240, December 1999.

[17] W. A. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Book Company, 1981.