Bachelor Thesis

A Graph-Based Approach To Access Control Migration

Author: Niklas Hackelberg
First reviewer: Prof. Dr.rer.nat. Chris Brzuska
Second reviewer: M.Sc. Estuardo Alpirez Bock

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for IT-Security Analysis
Hamburg, Germany

March 5, 2018

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die vorliegende Abschlussarbeit selbstständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt

Hamburg, 5.3.2018 _____

Niklas Hackelberg

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Access control is a fundamental part of information security, protecting the resources and data of systems against unauthorized changes, disclosure and use by regulating the actions a given user can perform. These regulations are usually described as some form of access control policy and are implemented in an access control mechanism. Often times it becomes necessary to migrate the implementation of a policy from one mechanism to another, be it due to new regulations, technological advancements and upgrades or switches to both new systems and access control mechanism or any number of other reasons. It is vital that during this migration process no user gets more or less access granted than before the migration. A manual migration is frequently not feasible due to the numbers of users, resources and rights, which is why there exist many access control migration tools. These tools however usually only handle a specific one-way migration with a fixed source and target mechanism and offer no comparison between the involved mechanisms.

The approach to access control migration used in this thesis is the creation of a graph-based access control model as an intermediate step in the migration process. The graph-model designed to be a common model of all involved mechanisms, enabling access control to be migrated from the graph to another mechanism, independent of which mechanism a graph was created from. This allows for the migration between any mechanisms represented by a given graph-model, but also for the comparison of the access granted by the different mechanisms. Additionally the nature of graphs also gives a visualization of the policies implemented by the mechanisms. The graphs in question will be created and modified by graph rules, which constrain all possible graphs to the common model and offer uniform transformations of the graphs. These rules can be reasoned about and they also simplify the creation of a graph to a series of rule applications for a given access control mechanism. The migration between access control in Db2 and access control in RACF serves as an example of the approach.

The thesis is structured into six Chapters and the structure loosely follows the steps of the graph based approach, by defining a common model between the mechanism, describing and reasoning about the derivations to an from the graph model and lastly implementing the derivations. Chapter 2 introduces background information about access control and graphs and the rule based transformation of graphs based on [Cor+96; Ehr+97; SS94; KKK06; KMP02; KMP05]. Chapter 3 gives an overview

of the two mechanisms described in [IBM17a; IBM17b; IBM17c], which is necessary information to create a common graph representation. Following that Chapter 4 is split into two parts. The first part details extensions to graph transformations and a graph-based model of the access control mechanisms described in Chapter 3. The access control models are similar to those described in [KMP05], but are adapted to the concrete mechanisms. The extensions to graph transformations are partly from [BFG94] and [KMP05]. The second part of Chapter 4 describes a derivation to and from a graph for each of the mechanisms. Since the derivations are independent of the other mechanisms involved, they allow for the migration of access control via the intermediate graph. The implementation of the model and algorithms introduced in Chapter 4 is described in Chapter 5. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background

This Chapter introduces some background knowledge necessary to understand the graph-model for access control and how the model is created and manipulated. The first Section describes some common terminology and concepts of access control. The second Section introduces the notion of graphs, graph morphisms and graph transformation rules used in this thesis.

2.1 Access Control

Access control is the management of access from legitimate users to the resources of a system, by denying or granting actions on a resource. Access control has to answer the question who is allowed to perform which action on which resource[SS94].

Usually users are the entities that try to access a resource. However user do not directly access resources, instead an access is made by a process. Processes are usually identified by some user identity or identifier. The process attempting to an access for the user is also called a *subject*[SS94]. Subjects can be granted or denied access to an object.

Objects are the resources of a system that are to be protected. These could be files, programs or any other identifiable resources of a system. Objects themselves can also be subjects[SS94]. For example an executable program is an object, because access to it can be regulated, but if the program makes accesses on behalf of a user, it also acts as a subject.

Access control *policies* are a high-level set of rules or privileges of what a user is allowed to perform which action on an object. These policies are then implemented by an access control *mechanism*. Mechanisms are software and hardware functions that implement the rules defined in the policy. One kind of policies are the *discretionary policies*(DAC), which define access control based on a user's identity and rules which define what a user is allowed to do or denied to do. DAC often additionally also allow the assignment of rules to groups. Groups are collections of users[SS94].

One common implementation of a DAC is the *access control list*(ACL), which are lists associated with objects that contain subjects and their allowed actions on the

associated object[SS94].

2.2 Graphs and Graph Transformations

2.2.1 Graphs

A *directed, labelled multi-graph* is a system $G = (V, E, s, t, l)$, where V denotes a set of vertices, E a set of edges, s a *source* and t a *target* function: $s, t : E \to V$ and lastly a label function $l : V \cup E \to \Sigma$, where Σ is a set of labels. All components of a graph G, can also be denoted as V_G, E_G, s_G, t_G, l_G.

A *path* is a sequence of edges $(e_1,\ e_2, \ldots,\ e_n)$, denoted as $e_1 \to e_2 \to \cdots \to e_n$, between two vertices a and b, such that $s(e_1) = a,\ t(e_n) = b, \forall i \in [1, ..., n-1] :\ t(e_i) = s(e_{i+1})$. If the number of edges between to vertices is not important, a path between a and b can also be denoted as $a \xrightarrow{*} b$

A graph G is a *subgraph* of graph H, if $V_G \subseteq V_H, E_G \subseteq E_H$ and $\forall e \in E_G :\ s_G(e) = s_H(e),\ t_G(e) = t_H(e)$. A subgraph is denoted as $G \subseteq H$.
Removal of vertices and edges can construct a subgraph, if whenever a vertex is removed, all its *incident edges* are also removed. Incident edges of vertex a are all edges e with $s(e) = a$ or $t(e) = a$. This guarantees that there are no edges whose source or target vertex does not exist, those edges are also called dangling edges. This means that the removal of $X = (V_X, E_X) \subseteq (V_H, E_H)$ results in a subgraph $G = (V_H - V_X, E_H - E_X, s, t, l) \subseteq H$, iff $\forall e \in E_H - E_X :\ s(e) \notin V_X,\ t(e) \notin V_X$ and $s(e) = s_H(e),\ t(e) = t_H(e)$ and $l(e) = l_H(e)$. In this case X fulfils the *contact condition* of X in H[KKK06].

Instead of removing edges and vertices, a graph G can also be extended with new vertices V_+ and edges E_+. The *extension* of a graph G with $(V_G, E_G, s_g, t_g, l_g)$ to a graph H is constructed with the structure $T_+ = (V_+, E_+, s_+, t_+, l_+)$, which does not have to be a graph. The source and target mappings $s_+, t_+ : E_+ \to V_G \sqcup V_+$ can also map to V_G and can contain an edge connected to a vertex that is not in its set of vertices. The symbol $\sqcup$ denotes the disjoint union of sets. H is constructed as $G + T_+ = (V_G \sqcup V_+, E_G \sqcup E_+, s_H, t_H, l_H)$ with $\forall e \in E_+ : s_H(e) = s_+(e), t_H(e) = t_+(e)$, otherwise $s_H(e) = s_G(e), t_H(e) = t_G(e)$[KKK06].

2.2.2 Graph Morphism

A *total graph morphism* f is a pair of total mappings $f_v : V_G \to V_H$ and $f_e : E_G \to E_H$ between two graphs G and H, with the mappings preserving the graph structure and the labelling, i.e. $\forall e \in E_G, \forall v \in V_G :\ f_v(s_G(e)) = s_H(f_e(e)),\ f_v(t_G(e)) = t_H(f_e(e)),\ l_G(e) = l_H(f_e(e))$ and $l_G(v) = l_H(f_v(v))$. [KKK06]

The image of a graph morphism $f : G \to H$ is called a *match* of G in H and is a subgraph $f(G) \subseteq H$, since the morphism is structure preserving and such fulfils the

contact condition[KKK06].

A *type graph* is a special graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$, that represents the type of vertices, edges and connections that may occur in a graph. A graph G is *typed over* TG, when there exists a pair $(G, type_{TG})$, where $type_{TG} : G \to TG$ is a total graph morphism. Type Graphs act as a constraint on graphs, since for example if there is no edge between two types of vertices in TG, then no graph typed over TG can have an edge there[KMP05]. A *typed morphism*, is a morphism $f : G \to H$ between $(G, type_1)$ and $(H, type_2)$, such that $\forall v, e \in E_G, v \in V_G : type_1(e) = type_2(f(e))$ and $type_1(v) = type_2(f(v))$[KMP05].

This chapter uses Pac-Man as an example for Graphs and Graph Transformations, the example loosely follows the example in [Ehr+97]. The type graph of the Pac-Man example can be seen in figure 2.1. It depicts the type graph for a graph model of the Pac-Man game. There are four types of vertices: a yellow Pac-Man-vertex, a blue ghost vertex, a pink Pac-Dot vertex and a black field vertex. The allowed edges are edges between fields and edges from Pac-Man, a ghost or a Pac-Dot to a field. Any graph G that is typed over this type graph has to follow this pattern. The edges between fields model the maze of the Pac-Man game and the edge between the other vertex types and a field model the placement of that vertex type on the maze.

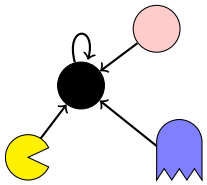


Figure 2.1: Type Graph for Pac-Man

2.2.3 Graph Transformation Rule

Graph transformation offers a rule-based method to change graphs, similar to formal grammars, but on graphs instead of alphabets.

A (typed) *rule* $r = (L \supseteq K \subseteq R)$, consists of three (typed) graphs L, K, R. The graphs L, K, R are called *left-hand side, glueing graph and right-hand side*. The basic idea behind a rule is that a graph G has to contain L for a rule to be applicable, R defines how G is changed locally, by replacing the match of L with R, and lastly K is used to describe how R is connected to G.

Rule Application

The application of a rule involves finding the match of the left-hand side in a graph G, then removing all parts that are in the match and not in the right-hand side, such that a match for the glueing graph can still be found in the replaced area. The graph G to which a rule is applied is also called the host graph[KKK06].

The first step in applying a (typed) rule r to a (typed) graph G requires a (typed) morphism $f : L \to G$ to establish a match of L in G, such that the following conditions are met[KKK06; Cor+96]:

- Contact condition of $X_r = (f(V_L) - f(V_K), f(E_L) - f(E_K))$ in G, i.e. removing the difference between L and K from G does not result in any dangling edges
- *Identification condition*: $f(x) = f(y)$ with x, y being either an edge or a vertex in L, iff $x = y$ or $x, y \in K$, i.e. f has to be injective for all elements in X_r

The contact condition ensures that removing X_r from G results in a subgraph. The identification condition prevents the conflict situation, where two vertices or edges x, y of L are mapped to the same vertex or edge in G and y is being deleted. In that situation it would not be defined if $f(x) = f(y)$ has to be deleted or not.

The second step is the creation of an intermediate (typed) graph $Z = (V_G - (f(V_L) - f(V_K)), E_G - f(E_L) - f(E_K))$. The graph Z is a subgraph of G, obtained by removing X_r, which is why the contact condition was necessary in the first step. As a subgraph of G the mappings of Z are defined as $\forall e \in E_Z : s_Z(e) = s_G(e)$ and the same for t_Z and l_Z respectively[KKK06].

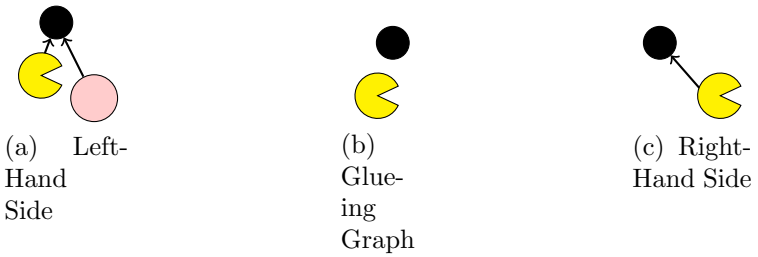
The third and last step creates the result (typed) graph H, by extending Z with the difference of R and K. H is defined as $H = Z + (V_R - V_K, E_R - E_K, s_{R-K}, t_{R-K}, l_{R-K})$. Since Z was created by removing the parts of L that are not in K, K still has a match $f(K)$ in Z. Therefore only the remaining vertices and edges of R need to be added. s_{R-K} is defined as $s_{R-K}(e) = s_R(e)$ if $s_R(e) \in V_R - V_K$ otherwise as $s_{R-K} = f(s_R(e))$ and the target and labelling function the same way respectively. This means that all new edges are connected to either a newly created vertex or to the match of K[KKK06].

After following all steps, the application achieved that every vertex and edge in the match of $L - K$ was removed, every one in $R - K$ was newly created, and all parts of K were kept and used to connect the newly created objects.

An example of a rule can be seen in figure 2.2. The rule 'Eat' depicts Pac-Man consuming a Pac-Dot. This rule is only applicable if Pac-Man and the Pac-Dot have an edge to the same vertex, i.e. are standing on the same location in the maze. The rule itself deletes the Pac-Dot and does not add any new elements to the graph to which it is applied.

Usually the glueing graph can be omitted as long as all vertices and edges can be uniquely identified in both L and R[KKK06]. In later depictions of rules, the glueing graph will be omitted and instead a rule will be denoted as $L \to R$.

An application of a rule r on graph G is called a *derivation*, denoted with $G \underset{r}{\Rightarrow} H$, where H is the resulting graph. A single derivation is called a direct derivation, while the iteration of those from $G_0 \underset{r_0}{\Rightarrow} G_1 \underset{r_1}{\Rightarrow} \ldots \underset{r_n}{\Rightarrow} G_n$ are called a *derivation* from G_0 to G_n. The sequence formed by $(r_0, \ldots, r_n)$ is called an *application sequence*[KKK06].

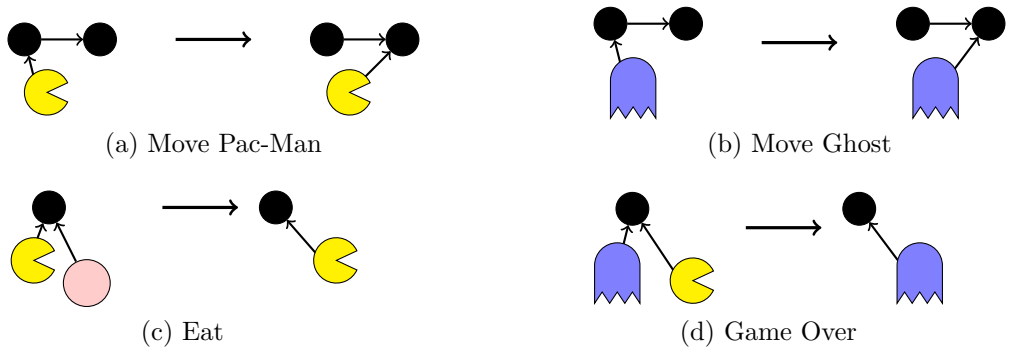


(a) Left-Hand Side (b) Glue-ing Graph (c) Right-Hand Side

Figure 2.2: Rule Example: Eat

Example

The figure 2.3 shows all rules of the Pac-Man example, typed over the type graph shown in figure 2.1. As mentioned earlier, the glueing graph is omitted.



(a) Move Pac-Man (b) Move Ghost (c) Eat (d) Game Over

Figure 2.3: Pac-Man Graph Transformation Rules

The figure 2.4 shows a derivation with the application sequence (Move Pac-Man, Eat, Move Ghost, Game Over). The match of a rule is expressed with a grey background. Note that at any point in the derivation multiple rules are applicable. Similar to formal grammars, graph rules and the graph grammars, that can be constructed with them, do not specify a rule order or a specific match a rule has to be applied too, if there are multiple matches.

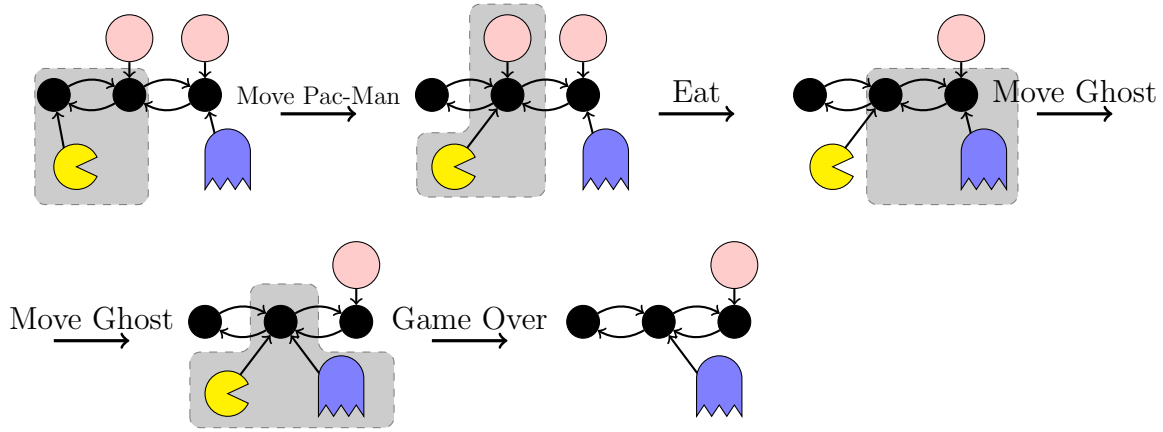


Figure 2.4: Example Derivation Sequence

The concepts of graph transformation rules introduced here, will be used to migrate access control, by deriving and modifying a graph with rules and type graphs that model access control. Later chapters will introduce extensions to rules, such as parameters, which will make them more applicable to a concrete problem, but also distances them from the theory of graph transformations.

Chapter 3

Access Control Mechanisms in z/OS

This Chapter introduces the two access control mechanisms that will serve as an example for the graph-based approach. The first mechanism is access control implemented by the database management system Db2 for z/OS[1]. The second mechanism is the RACF component of the z/OS Security Server, which is a layer of the operating system that can be called by other programs to resolve access control decisions. After describing the general concepts and behaviour of both mechanism the Chapter will finish with a description of how Db2 resources can be protected by RACF and details the difference between the two mechanism. This information is necessary to build the common graph-model of the two mechanisms in the next Chapter.

3.1 Db2 Access Control

Db2 controls access to its objects, based on *authorization identifiers(IDs)* and the *privileges* and *authorities* granted to an ID. Each *Db2 object* has an associated set of privileges. The table A.1 shows an overview of all Db2 objects that have a set of explicit privileges and table A.2 shows as an example of the privilege set for the specific objects Table and View. Db2 offers multiple ways for a subject to obtain its IDs as well as its privileges, which will be described in the next sections.

3.1.1 Authorization IDs

Each subject connecting to Db2 is associated with at least one identifier. The *primary authorization ID* is the identifier used to identify a process and depends on the way a user connected to Db2. This ID will usually be the ID used during login. All other IDs are considered to be *secondary authorization IDs*, these could be for example an ID associated with a RACF group. These IDs can hold additional privileges of the subject. IDs are set during the connection and sign-on routines, these routines can be executed outside of Db2 and introduce new IDs, like the mentioned RACF groups[IBM17a].

[1]Operating system for IBM mainframes

Furthermore there is one SQL ID and one RACF ID, these IDs can be set to either the primary or a secondary ID. The SQL ID is used for some dynamic SQL statements, the RACF ID is used for multilevel security or when using the RACF Access Control Module[IBM17a; IBM17b].

3.1.2 Privileges and authorities

Privileges and authorities allow an ID to perform an action on an object. There are two types of privileges: *explicit privileges* and *implicit privileges*. Furthermore there are *administrative authorities*. While Db2 privileges are not hierarchical, authorities are[IBM17a].

Explicit privileges are privileges that can be granted and revoked with GRANT and REVOKE SQL statements. Each privilege has a specific name and set of actions that are allowed when one has been granted the privilege. Explicit privileges can be granted to a specific ID or to the *PUBLIC ID*. The latter one grants the privilege to all IDs known to Db2[IBM17a]. The privilege information is stored in the *Db2 catalog* and can be fetched by querying the tables *SYSIBM.SYSXAUTH*, where is X replaced with a Db2 object type. All relevant catalog tables are listed in the table A.3.

There are two types of implicit privileges. One type are privileges granted by ownership of objects. These privileges only extend to the owned object itself. Another type of implicit privileges are the ones obtained by executing a package[2] and plan[3]. Subjects that have the EXECUTE privilege on them can start those, without needing to have the necessary privileges for all action the plan and package will perform. This way the subject also implicitly is granted all authorities granted to the package or plan. Alternatively the package or plan can only be executed if the subject already is granted all privileges used by them. This behaviour is set by the owner of the plan or package.

Authorities are sets of privileges, which are necessary to fulfil certain roles in administrating a database, such as database admin (DBADM). Many of these privileges can not be explicitly granted and can only be obtained by being granted an authority. There are Db2 system authorities, like SYSADM, and there are Db2 database authorities, like DBADM[IBM17a].

With all these different privileges, there are multiple ways to met the requirements necessary to be authorized to perform an action. For example to be able to insert a value in a user defined table, the user has to met at least one of the following conditions: INSERT privilege on the table, ownership of the table, or assignment to either the DBADM, SYSADM or DATAACCESS authority.

[2]Control structure containing executable SQL statements
[3]Application plan containing a list of packages

3.1.3 Subsystem access control

Db2 also offers the option to authorize actions from outside of the Db2 subsystem[4]. For this Db2 defines exit-points to its connection, sign-on and authorization routines. Then outside routines like RACF or custom ones can resolve authorization and return whether authorization was granted, denied or if Db2 should use its own authorization checking. In combination with RACF, Db2 can offer many more access control concepts like *Mandatory Access Control*, hierarchical groups and row-level granularity. However this thesis uses the two access control mechanisms as examples for migration and as such only considers exclusive use of one mechanism or the other[IBM17a; IBM17b].

3.2 RACF

The *Resource Access Control Facility(RACF)* is a component of the z/OS Security Server, that offers centralized user verification, authentication, authorization and logging. RACF grants access, to the resources, it is protecting, according to ACLs defined on these resources. RACF uses its own database to store the access control information, grouped as so called *profiles*. There are different types profiles namely profiles for users, groups, data sets[5] and general resource profiles[IBM17c]. From now on the term RACF profile without a qualifier like user or group is used to refer to RACF general resource profiles.

RACF does not actively prohibit access to a resource. Instead when a user tries to access a resource from for example a terminal or from the Db2 database, those subsystems consult RACF and supply it with a set of parameters. RACF then performs a lookup in its database and checks if the given user is in the ACL of the RACF profile that is protecting the given resource and action. Then RACF returns a value, stating the access is allowed, forbidden or it can not supply an answer and the calling subsystem should resolve the access itself.

3.2.1 RACF users and groups

Access in RACF can be granted to users, groups and or to all users and groups. The latter case is defined by the *universal access authority*(UACC)[IBM17c].
RACF Users are defined in a user profile. A user profile consists of a base segmented and multiple optional segments. The base segment contains fields for the user's identifcation (USERID), fields relating to passwords, the profile's owner, the user's default group and multiple fields like security labels, RACF privileges and public keys, which are used by other security mechanisms[IBM17c].
A RACF group is a collection of users, defined in a RACF group profile. Each RACF user has a default group and is a member of at least that group. The group profile also consists of multiple segments, a base one and several optional ones. The base segment contains the group's name, the group's owner and the group's superior group (SUPGROUP). With the group's owner and superior group the RACF groups are organized

[4]System processes, service provider that are inactive until a request is made
[5]z/OS file with a record structure

in a tree hierarchy. However, the hierarchy is not used inheritance of access rights, but for RACF administration. Each member of a group can access resources with the authorities assigned to the group. Members of a group are said to be *connected* to the group[IBM17c].

3.2.2 Resource protection

The ACLs that RACF uses to control access are stored in resource profiles. Those profiles are identified by their *class* and profile name. The class stands for the kind of resource the profile is protecting, more on that in section 3.3. A resource is protected by a profile, if the resource name and classes match the one of the profile. Resource names are composed of qualifiers separated by a dot (.) and are generated by the subsystems that query RACF. There is no direct connection between the actual resource and its profile. A profile can be created before a resource exists and can persist when a resource is deleted. Furthermore there are profiles that match multiple resources or group different resources together[IBM17c]. Resource profiles can be generic or distinct. Generic profiles can contain *generic characters*, such as asterisks (*), double asterisks (**), percent-signs (%) and ampersand (&) which act as wildcards or variables during the matching between a resource and its corresponding profile. The two types of asterisks have multiple behaviours depending on their position in a profile name and if they are the only character in a qualifier or not. The behaviour ranges from matching any one character to all characters and matching any qualifier to all qualifiers. Furthermore the % character matches any single character in a profile name. The ampersand denotes a variable, which can be set and changed separately from the profile and will be evaluated during resource access. Due to the wildcards multiple profiles can match for one resource. In this case the ACL of the least generic profile is used to determine if access is allowed or not. To be less generic than other profile means either being discrete or having a generic character further right in the profile name. There are also rules for resolving conflicts when two profiles have a generic character in the same position. The table A.4 shows an excerpt from an example from [IBM17c], detailing which profile is matched with which accessed resource. The profile names are order from discrete over least generic to most generic[IBM17c].

Another way of controlling access to multiple resources with one profile is the use of *grouping/group profiles*. These are resource profiles where the name does not match with the resources they are protecting. For this to work the resources class has to be a *grouping class*. If that is the case then grouping profiles can be created for resources of this class. Grouping profiles contain a list of members, which are the actual members to which access is controlled[IBM17c], so the profiles that the resources are matched against.

Each of the resource profiles mentioned here also contain an ACL which mention what *access authority* a user or group has on the profile. Profiles are created by a *RDEFINE classname profilename* command. The access authorities or privileges that RACF offers are NONE, READ, UPDATE, CONTROL and ALTER. These privileges are hierarchical, so a user who has the permission to ALTER a resource also has all

other permissions. The privileges are granted with a *PERMIT* command. RACF started out as access control for datasets and as such the permissions were named after operations on files. However, since RACF can protect more abstract resources, these permissions do not intuitively translate to resource specific actions[IBM17c].

All the parts of RACF mentioned here are just a subset of what RACF offers, but this is the subset necessary to use and migrate Db2 access control.

3.3 Db2 with RACF

As mentioned in Section 3.1 Db2 offers an exit point to allow for authorization from outside of the subsystem. One possible module for the exit point is the RACF access control module, which uses RACF for authorization. The module translates a Db2 access to various parameters, which it then supplies to RACF. However, for this module to work all concepts of Db2 access control have to be mapped to concepts of RACF[IBM17b].

Each Db2 object has its own RACF class and grouping class, depicted in table A.5. It is not possible to create a single RACF profile for one Db2 object, since RACF has a fixed set of hierarchical permissions and each Db2 object has its own set of non-hierarchical privileges. There is no direct mapping between the two. Instead each explicit privilege granted to an object is a resource that can be protected by a RACF profile. The resource has the form *SUBSYSTEM.OBJECTID.PRIVILEGE*. This resource can be protected by a RACF profile as mentioned in Section 3.2. So, when a user has the privilege to perform a SELECT on the table TEST in the schema SYS in the Db2 subsystem DB, then this privilege expressed as a RACF profile means the user is in the ACL of the RACF profile that protects the resource DB.SYS.TEST.SELECT with at least a READ permission. The RACF profile in this example is either in the class MDSNTB or a member of a GDSNTB grouping class. When a user has the ALTER privilege on a RACF profile this is effectively the same as being granted the corresponding explicit privilege with the option to grant it to others. The assignment of authorities works similar, each authority has its own resource that can be protected by a RACF profile[IBM17b].

The implicit privileges in Db2 do not have to be migrated, since they have no representation in RACF and are always resolved by Db2 itself. Ownership behaves differently in RACF and in Db2. While in Db2 ownership grants certain implicit privileges, these can not be expressed by ownership of a RACF resource profile or a RACF group/user profile. The RACF access control module can check these implicit privileges itself, since it already constructs parameters containing an object's owner and the accessing user's id for RACF. So the implicit ownership can be resolved with the supplied parameters without accessing RACF and its database[IBM17b; IBM17c]. This means that ownership is not a value that can be migrated nor does it have to be migrated. Implicit privileges gained by executing packages are also resolved by Db2, since they do not correspond to a specific privilege and in the case of dynamic SQL are only resolved at runtime. But the privilege to execute a package is a resource that

can be protected by RACF.

The other privileges however can be represented in RACF and behave exactly the same to Db2. To use the example from section 3.1 a user is authorized by RACF to insert a value in a user defined table, if the user mets one of the following conditions: ownership of the table, READ or higher access to the resource profile that protects the resource *SUBSYSTEM.TABLE.INSERT* or to the profiles that protect the resources of the administrative authorities DBADM, SYSADM, DATAACCESS.

Chapter 4

Access Control Migration

This chapter will introduce a graph-based model for access control, which will be used to migrate access control between the mechanisms presented in Chapter 3. The model will be a common representation of all mechanisms involved and each of them has to define a derivation to and from the model. This way access control can be migrated between every modelled mechanism.

Both mechanisms used here have the same notion of objects, actions on these objects and when an action is authorized, i.e. the same privileges are necessary for the authorization of an action. Therefore the common model has to represent the actual privileges, and the entities who they apply to. This will simplify the common model as well as the derivations shown in sections 4.2 and 4.3, which define how a graph is derived from a mechanism and a mechanism derived from the graph. The model presented here can be used for any mechanism that has the same notion of when access is granted.

However, this graph-based approach would also work for mechanisms that behave differently, as long as one can find a common model for all involved mechanisms. An example could be two mechanisms, that have different sets of conditions, that have to be satisfied for an action to be authorized. Then a common model might only represent the intersection of both conditions or just models the actual actions that can be authorized. Depending on the accuracy used in the common model a migration might grant more or less than it did before. These considerations have to be weighted against each other. In the case of an intersection, the migration might only capture a subset of the policy.

A model might only show that a user is authorized to perform an insert on a table. If the instance of the model was created from the Db2 privileges, then the model shows this authorization if the user is assigned the explicit privilege, ownership or one of the necessary authorities. Such a model no longer shows exactly which privilege granted this authorization. Therefore a mechanism that wants to implement the model has to use one of its ways to authorize this action. The model shown here has the objects and the privileges on them as a common element of the two mechanisms and thus models those.

4.1 Graph-Based Access Control

4.1.1 Extensions to Graph Transformation Rules

Before being able to use the graph rules to create a model of access control privileges some extension to the concept of rules have to be made. Without them there is no way to specify which rule is supposed to be applied at a given event, on which match a rule is applied to or control the labels created by a rule. While it would be possible to achieve such behaviour by creating a rule for all possible label combinations, this would be quite cumbersome. Instead rule parameters will be introduced as well as event driven application and further constrains on the application of a rule. These changes increase expressiveness, however they move the rule concept away from theory of graph transformations.

To further constrain when a rule is applicable, a *negative application condition*(NAC) is introduced. A (typed) negative application condition is a pair of (typed) graphs (L, N) with $L \subset N$. N denotes a graph that may not appear in a graph for a rule to be applicable. This means that the match of L in a graph G can not be extended to form a match of N. From now on the left-hand side of a rule will show N, with $N - L$ being drawn with dotted lines and L being drawn with solid ones[KMP02; BFG94; KMP05].

Extending rules to contain a set of parameters allows to constrain when a rule is applicable, but also which labels the newly created vertices and edges have [BFG94]. Figure 4.2 depicts the rules for Db2 access control. The labels in the left- and right-hand side graphs of each rule contain place-holders x, y and z, which are also found in the parameter list of the rules. Any value given to a rule , replaces its place-holder in the label function of the left- and right-hand side graph, before the rule is applied.

There are various approaches to order rule application, such as assigning each rule a priority and all higher priority rules have to be applied before a lower priority rule can be applied. However in this case the rule order is determined by the information of access control mechanisms. This approach is called *event-driven graph transformation*[BFG94], since the order of rule application is based on events of an external system.

An example of a rule application of an extended rule would be invoking the rule *Add User(Alice)*, shown in figure 4.2. The label Ux will be replaced by $UAlice$ and the rule itself is only applicable, if a host graph G does not contain a vertex x with $l_G(x) = UAlice$. Since the vertex of the left-hand side is drawn with dotted lines and as such can not be a part of the host graph. This guarantees that the rule will only create unique users.

4.1.2 Modeling Db2 ID-Based Access Control

The privileges of the access control mechanism of Db2 will be modelled as a graph typed over the type graph presented in figure 4.1. The type graph provides the types U for users, O for Db2 objects and A for administrative authorities. Explicit privileges are represented by an edge from an user to an object labelled with the name of the

privilege. Assignment of authorities is represented by an edge from an user to the authorities the user is assigned to. The type morphism $type_{DB2}$ for any graph G, that models Db2, maps all vertices and edges with label Tx to the type T in the type graph, i.e. a vertex with label $UBob$ gets mapped to the vertex with label U in the type graph.

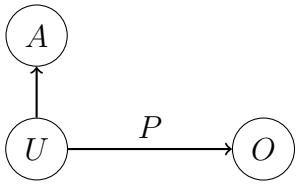
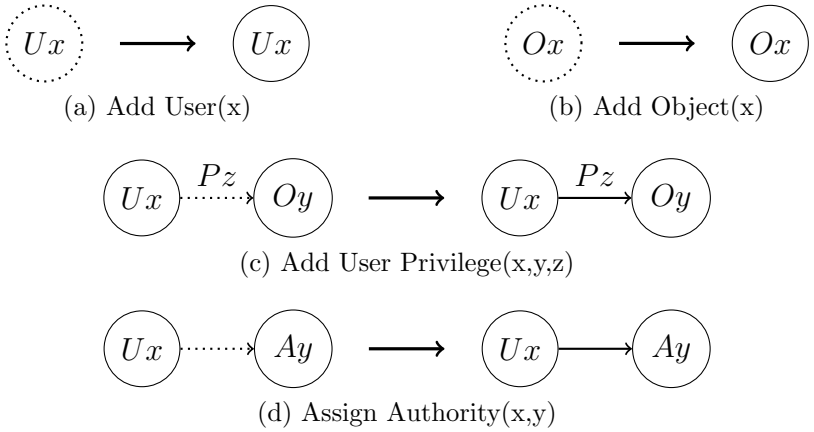


Figure 4.1: Type Graph for Db2 ID-Based Access Control

The rules used to create and manipulate a graph G, that represents Db2 access control, are shown in figure 4.2. As mentioned earlier, the rules use negative application conditions and parameters. An example of how such a rule works was given in Section 4.1.1. The NAC ensure that all users, objects and privileges are created and assigned at most once, since all rules can only add a vertex or an edge if that object with the given label is not already part of the host graph.



(a) Add User(x)

(b) Add Object(x)

(c) Add User Privilege(x,y,z)

(d) Assign Authority(x,y)

Figure 4.2: Graph rules for Db2 ID-Based Access Control

In this model an explicit privilege p is granted on object o to user u, if the modelling graph G has some edge $e : l(s(e)) = u,\ l(t(e)) = o,\ l(e) = p$. An authority a is assigned to u, iff $\exists e \in E_G : l(s(e)) = u,\ l(t(e)) = a$.

Since the authorities are fixed for Db2 there is no need for a rule to create authorities. The host graph that will be modified by the rules will just contain all vertices for the authorities from the start.

4.1.3 Extended Model

To be able to also model access control to Db2 with RACF the type graph and rules have to be extended to the type graph shown in figure 4.3 and the rules shown in figure

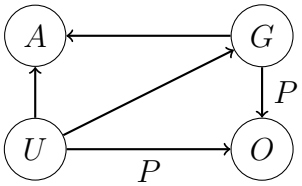


Figure 4.3: Type Graph for Db2 with RACF Access Control

4.4. It's important to note that the objects and the privileges are not RACF profiles and their hierarchical privileges, but the Db2 objects and their respective privileges. This makes the derivation to and from RACF a bit more complex, but otherwise the model would not be a common model between the two mechanisms. A RACF profile with a READ access is not a concept used in Db2. Furthermore, this also means that any general profile has to be expanded and turned into all Db2 objects and privileges it is protecting.

Additionally to the old type graph's vertices and edges and their representations the new type graph includes the vertice G, which represents a group. Each group vertex can be assigned a privilege P on an object, represented by an edge, just like a user vertex and the same for administrative authorities respectively. Membership of a group is represented by an edge between a user and a group.

The extended rules encompass all rules of of the Db2 ID-based Access Control with the Add Privilege rule being renamed to Add User Privilege. Furthermore, there are new rules to create groups and to connect users to a group. The rule Add Group creates a new group with label x, if the group does not already exist.

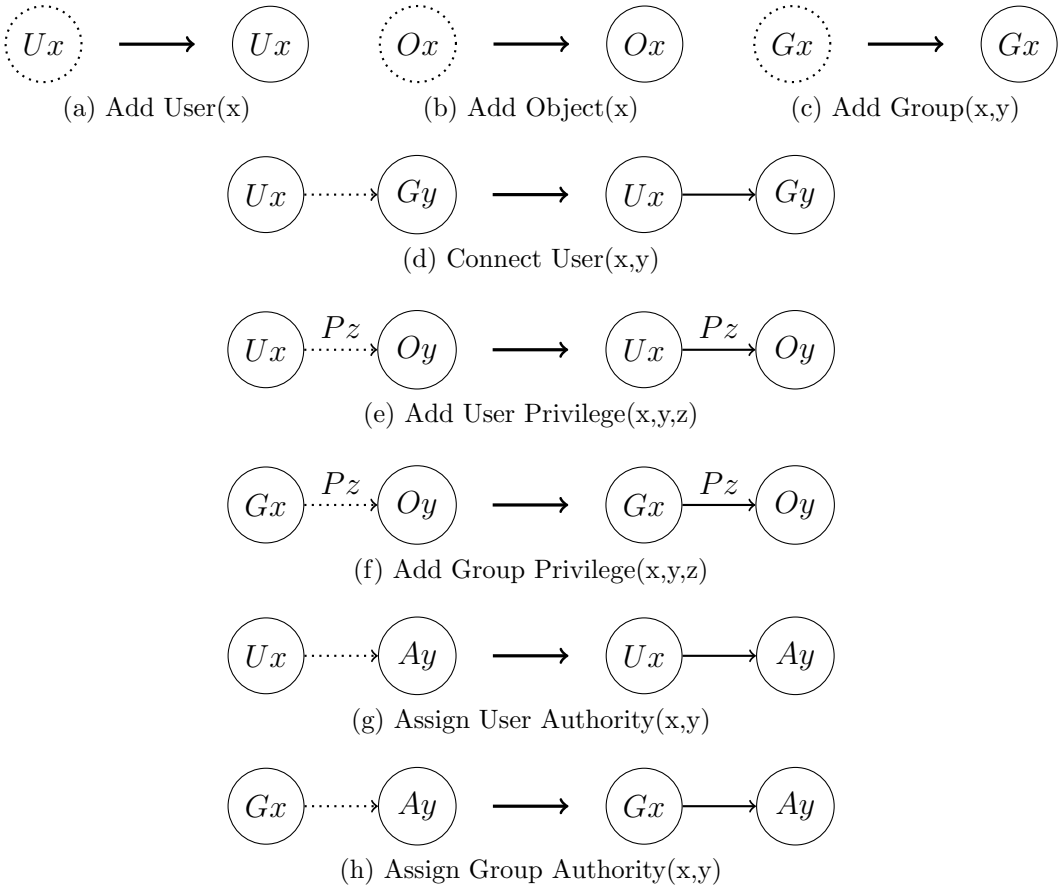
The next Sections of this Chapter detail the derivations from the and to the graph model described here. An example of these derivations and the graph model can be found in the Appendix under Chapter B.

4.2 Graph Derivation from Db2

The derivation from and to Db2 has to take into account that Db2 has less features than RACF. This for example means that, instead of simply looking for a single edge between an user and an object or authorities, the derivation has to look for paths between them, since a user could have obtained a privilege over group membership.

4.2.1 Db2 to Graph

As mentioned in Section 3.1 Db2 privilege information is stored in the catalog tables SYSIBM.SYSXAUTH, where X is a Db2 object. An overview of the catalog tables can be seen in table A.3. The general structure of such a table is depicted in table 4.1. A table contains a GRANTEE column, which contains the authorization ID all privileges of a row are granted to. Furthermore a table contains some columns that hold values



(a) Add User(x) (b) Add Object(x) (c) Add Group(x,y)

(d) Connect User(x,y)

(e) Add User Privilege(x,y,z)

(f) Add Group Privilege(x,y,z)

(g) Assign User Authority(x,y)

(h) Assign Group Authority(x,y)

Figure 4.4: Graph rules for the Extended Model

identifying an object and lastly multiple columns for privileges on the object. These privileges are one character columns, which can either be 'N','Y' or 'G' for no privilege, privilege granted or granted and can be granted to other users respectively. For example the catalog table for Tables and Views SYSIBM.SYSTABLEAUTH contains rows for the privileges depicted in table A.2.

OBJ.:	GRANTEE	OBJECTID	P1AUTH	P2AUTH	...	PNAUTH
	⋮	⋮	⋮	⋮	⋮	⋮
	Alice	DEMO	Y	N	...	G
	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.1: Sketch of an authorization catalog table

The algorithm 1 shows pseudo-code, which for a given host graph G, an Db2 object type T and the corresponding authorization catalog table A adds all explicit privileges granted for all objects in T. For each row the algorithm attempts to invoke the rules to add the user, add the object and add all privileges the user has on the object. Objects are both labelled with their identifier, as well as their object type. Privileges are labelled with their name and additionally a '*', if the privilege was granted with the ability to grant it to others.

Algorithm 1 Add Explicit Privileges from Db2

```
1: INPUT: Graph G, ObjectType T, CatalogTable A
2: OUTPUT: Graph G with explicit privileges for Db2 Object T added
3: for Row r ∈ A do
4:     G.addUser(r.GRANTEE)
5:     ObjectLabel l = T + r.OBJECTID
6:     G.addObject(l)
7:     for Privilege p ∈ r do
8:         if (p.AUTH == Y) then
9:             G.addUserPrivilege(r.GRANTEE, ObjectLabel, p.NAME)
10:        else if (p.AUTH == G) then
11:            G.addUserPrivilege(r.GRANTEE, ObjectLabel, p.NAME + '*')
12:        end if
13:    end for
14: end for
15: return G
```

The assignment of administrative authorities works similar to the algorithm 1 with the difference being that Assign Authority is being invoked instead of Add User Privilege.

4.2.2 Graph to Db2

The algorithm 2 shows pseudo-code that creates SQL-Grant statements from a given graph model. Since the RACF mechanism allows for hierarchical groups, it is not enough to simply look for an edge between a user and an object, but to look for a path from a user to the object. The privilege that is granted with such a path based on the label of the last edge. Since a single edge already counts as a path, this algorithm works for both graphs created from the Db2 mechanism as well as from the RACF mechanism.

Algorithm 2 Create GRANT Statements

```
1: INPUT: Graph G
2: OUTPUT: SQL-Grants S for all explicit privileges in G
3: Users U = G.getUsers
4: Objects O = G.getObjects
5: SQL-Grants S = ∅
6: for User u ∈ U do
7:     for Object o ∈ O do
8:         for Path p : e_1 → ··· → e_n between u and o do
9:             S.add(GRANT l_G(e_n) ON l_G(o) TO l_G(u))
10:        end for
11:    end for
12: end for
13: return S
```

The assignment of administrative authorities works similarly. Each user with a path to an authority is assigned the authority.

4.3 Graph Derivation from RACF

4.3.1 RACF to Graph

The derivation from RACF to the common graph model is more complicated than all the other derivations, because RACF profiles do not have a direct connection with the objects they are protecting. This means that a profile can exist before and after a Db2 object exists and multiple profiles can match a single Db2 object.

The algorithm 3 shows the RACF equivalent of the algorithm 1. Since the RACF profiles have no information what actual Db2 objects exists, this algorithms still needs to query the Db2 catalog for all Db2 objects of a specific type. Furthermore a RACF profile does not necessarily contain the actual Db2 privilege in its name. This means the algorithm has to find the match for each Db2 object and all the possible privileges defined on that object. To achieve this, the RACF profiles are sorted from discrete over least generic to most generic in line three. Because of this order the first match found in the RACF profiles is the actual profile that protects that object and its explicit privilege. Once the profile is found, all subjects with at least READ authority on the matched profile can be added to the model and granted the privilege. For simplicity

the pseudo-code only shows the case where the subject is a user, but the algorithm has to differentiate between users and groups. All subjects with ALTER authority on the profile have to be added with the '*' mark, which signifies that the privilege is grantable to others.

Algorithm 3 Add Explicit Privileges from RACF

```
1: INPUT: Graph G, ObjectType T, CatalogTable A, RACF Profiles R in Class of T
2: OUTPUT: Graph G with explicit privileges for Db2 Object T added
3: R = R.sort()
4: Objects O = A.getObjects()
5: for Object o ∈ O do
6:     for Privilege p of o do
7:         RACF profile r = R.firstMatch(T.o.p)
8:         G.addObject(o)
9:         for Subject s ∈ ACL of r with s has at least READ authority do
10:            G.addUser(s)
11:            G.addUserPrivilege(s,o,p)
12:        end for
13:    end for
14: end for
15: return G
```

Since the RACF authorities are implemented as an ACL, there is no need to query all users. They are simply listed in the ACL. However the groups listed in the ACLs have to be resolved to all users connected to a group, which are then connected to a group with the Connect User rule.

The modelling of the administrative authorities is similar to the algorithm 3, with the difference that all administrative authorities are fixed and do not need to be queried. Instead the sorted RACF profiles can be searched for the profiles that define the administrative authorities and all users and groups in the ACL are assigned the administrative authority.

4.3.2 Graph to RACF

Since the object vertices of the graph do not represent RACF profiles, the derivation of RACF commands require more effort than the derivation of Db2 commands. The algorithm 4 creates the commands that define the RACF resource profiles. For every object vertex in the graph, the algorithm constructs a set of privileges that are granted on this object. These privileges are the labels of every edge that have the object as a target. The resulting profile names are the class of an object, the name of the object and the privilege granted on the object, exactly the discrete profile that grant a privilege as described in section 3.3.

The creation of groups and their membership is achieved by first defining all groups based on the group vertices of the graph and then each user with an edge to a group

Algorithm 4 Create RACF resource profiles

```
1: INPUT: Graph G
2: OUTPUT: RACF-Comamnds R to create profiles
3: Objects O = G.getObjects
4: RACF-Commands R = ∅
5: for Object o ∈ O do
6:     Set of Privileges P = ∅
7:     for Edge e : t_G(e) = o do
8:         P.add(l_G(e))
9:     end for
10:    for Privilege p ∈ P do
11:        R.add(RDEFINE l_G(o).p)
12:    end for
13: end for
14: return R
```

vertex will be connected to the group.

The RACF resource, group and user profiles have to be defined before the privileges can be assigned. Once these profiles exists, the privileges on them can be granted similarly to the algorithm 2, with the difference, that this time the algorithm does not look at paths with more edges than one. Instead it looks at the incident edges of each object and adds a PERMIT for the user or group as shown in algorithm 5. Every user or group that has an edge to an object, gets READ authority on the profile with the name of the object concatenated with the label of the last edge of the path. If the label of the edge is also marked with '*' the ALTER authority instead of the READ authority is granted, allowing the user or group to grant others access to the profile. Administrative authorities are similarly assigned if there is an edge between a user or a group to an authority.

Algorithm 5 Create RACF PERMIT Commands

```
1: INPUT: Graph G
2: OUTPUT: RACF Commands P for all explicit privileges in G
3: Objects O = G.getObjects
4: RACF-Commands P = ∅
5: for Object o ∈ O do
6:     for Edge e : t_G(e) = o do
7:         P.add(PERMIT l_G(o).l_G(e) ID(l_G(s_G(e))ACCESS(READ))
8:     end for
9: end for
10: return S
```

Chapter 5

Implementation

This Chapter describes the implementation of the concepts and models introduced in Chapters 2 and 4. First the structure of the implementation is described, detailing the important classes, their methods and how they relate to the introduced concepts. Afterwards the input and output of the program is described as well as the workflow. And the last section evaluates the issues and benefits of the implementation with respect to the specific mechanisms.

5.1 Structure

The project was implemented in the object-oriented language *Java* and is structured into three groups of classes. The first group is the *AccessControlGraph* class and its components, which implement the graph-based access control model and the graph transformations rules. The other groups are the interfaces *GraphBuilder* and *CommandWriter* together with the classes that implement them. Each class implementing an interface, is realizing one of the derivations described in the last chapter. The figures C.1 and C.2 show UML diagrams of all mentioned classes.

The class AccessControlGraph has a directed multi graph as a member, implemented with the graph library *JGraphT*, which allows for the creation of various different types of graphs and offers algorithms on these graphs, such as finding paths. The edges and vertices of the graph can be custom Java classes. In this case the vertices are a class hierarchy that hold a label identifying a vertex and the type of the vertex according to the type graph in figure 4.3. In the case of object vertices the Java object also holds the Db2 type of the object. The edges have members for their privilege and their source and target vertex. Furthermore, they have a boolean member detailing if they are grantable or not. This way, the labels that were described in Section 4.1.1 are turned into various members of the vertex and edge objects.

The graph rules defined in figure 4.4 are implemented as public methods of the AccessControlGraph class. These methods contain various control flow statements that model the graph morphism and rule application of Section 2.2 and the rule constraints of Section 4.1.1. An example of such a method can be seen in figure 5.1, which shows the implementation of the Add User rule. In the figure the member m_graph is the

mentioned directed multi graph.

Additionally the AccessControlGraph class offers methods that return sets of vertices grouped by their vertex type, so for example a set of all users, or certain subgraphs, like the subgraph formed by the group hierarchy. And it offers a method that returns all paths between two sets of vertices.

All these public methods of the AccessControlGraph class are invoked by the classes implementing the GraphBuilder interface, which is responsible for the derivation to a graph, or the CommandWriter interface, which is responsible for the derivation to a mechanism's commands. The derivations work like described in Sections 4.2 and 4.3. Each derivation is implemented in its own class. The classes are called by the interfaces they implement, this way they can be freely mixed and matched. Which results in the access control being migrated to any implemented mechanism regardless of the source mechanism. Furthermore the use of interfaces makes the addition of other mechanisms fairly easy.

```
public void addUser(String uid) {
    Vertex user = new Vertex (uid, VERTEXTYPE.USER);
    //L

    //L-N
    if (m_graph.containsVertex(user)) {
        return;
    }
    //R
    m_graph.addVertex(user);
}
```

Figure 5.1: Implementation of the AddUser Rule

5.2 Input and Output

The input of the program depends on the mechanisms that the program is supposed to migrate to and from. Therefore one set of input parameters has to specify which are the source and the target mechanism. For the cases of the mechanism implemented here both require the querying of Db2 catalog tables and thus parameters that can be used to establish a connection with the database. For deriving the graph from Db2, the privileges have to be queried and for deriving the graph from RACF the existing Db2 objects have to be correlated with the RACF profiles protecting them. Furthermore in the case of RACF the program has to be supplied with files containing the RACF profile information, since RACF does not offer a Java API that can query RACF resource profiles. Such a file can be generated with the script language REXX[1], which

[1]Restructured Extended Executor, a script language developed by IBM

can interface with RACF.

The output of the program is a file containing the target mechanism's commands, which were derived from the graph. These commands can then be entered as a batch job to the z/OS. Additionally the graph itself is also written to a separate file in the GraphML format[2], this could be later used to visualize the graph or compare it with other graphs.

5.3 Evaluation

Comparing the implementation of a policy before and after the migration shows that the migration changes how the policy is implemented depending on the source and target mechanisms. This change, in the case of RACF, also happens when source and target mechanism are the same. The change is partially caused of the choice of common model, but also by the fact that Db2 offers only a subset of what RACF offers.

Since the common model represents Db2 objects and the privileges on them, it does not represent generic RACF profiles or RACF authorities. Therefore the exact RACF authorities and RACF profiles can not be recreated from the graph. The behaviour of RACF authorities for Db2 resources effectively only has three cases, which can be recreated. All users or groups that have a privilege, get mapped to READ authority, all that have the privilege with the grantable option get mapped to the ALTER authority. Therefore the RACF authorities UPDATE and CONTROL are lost, but the behaviour stays the same. Generic RACF profiles can not be recreated since all RACF profiles are matched with the Db2 objects and privileges they are protecting in the derivation process. This results in no subject having more or less privileges on the Db2 objects, that existed at the time of migration. However, since RACF profiles can exist before the resource, they are protecting, exits and thus protect 'future' objects, this protection is lost after a migration. In the case of Db2 as a target mechanisms, the groups are lost, since Db2 has no equivalent concept to RACF groups on its own.

Regardless of the changes made to the implementation after a migration, the use of an intermediate model does allow for the mixing of source and target mechanism independently of each other. The derivations defined in Sections 4.2 and 4.3 result in a migration where each subject is authorized to no more or less actions on objects than they were before, with the limitations that this only applies to objects that exist at the time of the migration.

[2]XML-based file format for graphs

Chapter 6

Conclusion and Future Work

The use of an intermediate model during the migration process had the wanted quality of enabling migration without fixing source and target mechanism. However, both the quality of the migration, i.e. if privileges or concepts are lost or gained and how many mechanisms can be migrated to, does not only depend on the involved mechanisms any more. They also depend on the design of the common model. As mentioned in Section 5.3 the example model of Db2 and RACF already shows that generic profiles are lost due to the design of the common model and with them the privileges granted on 'future' objects. On the other hand the loss of the RACF groups when migrating to Db2 is a loss that is inevitable, due to the mechanism Db2 lacking a group concept.

One solution for this problem would be to expand the graph-model to not only include common elements of all involved mechanisms, but also elements only used in one or more. In the given example one could also introduce a node for RACF profiles and connect all Db2 objects and privileges to the RACF profile node that is protecting them. The profile nodes would simply be ignored in the derivation to Db2, but could be used in the case of a derivation to RACF to restore the original RACF profiles.

While not explicitly mentioned in this thesis, the creation of models also has benefits not directly related to access control migration. It allows for the uniform comparison of two or more mechanisms, by comparing the resulting graphs, instead of finding and querying comparable metrics for each mechanisms. For example, one could compare the privileges a user has, by looking at all paths that a user has to objects, and seeing if a user has more or less edges to objects in one or the other graph model. Or instead of deriving the graph model from an implementation, it could be created manually in a kind of editor and from that a given target implementation could be derived. The choice of graphs as a model has benefits that could be used to optimize or find faults in a given implementation. Graphs can be visualized and might offer some intuition as to where there are mistakes or where there is potential to improve an implementation of a policy. A more rigorous approach to optimization could be the analysis of graph networks. Metrics like modularity and the detection of clusters or communities could show opportunities to introduce groups or related concepts like roles[1].

[1]Collection of privileges

Bibliography

[BFG94] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. "Issues in the practical use of graph rewriting". In: *International Workshop on Graph Grammars and their Application to Computer Science*. Springer. 1994, pp. 38–55.

[Cor+96] A Corradini et al. "Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach". In: (1996).

[Ehr+97] Hartmut Ehrig et al. "Algebraic approaches to graph transformation: part ii: single pushout approach and comparison with double pushout approach". In: *Handbook of Graph Grammars*. 1997, pp. 247–312.

[IBM17a] IBM. *Managing Security*. Db2 12 for z/OS. SC27-8854-02. Last accessed on 2018-3-1. IBM Corporation. 2017. URL: `https://www.ibm.com/support/knowledgecenter/SSEPEK/pdf/db2z_12_secabook.pdf`.

[IBM17b] IBM. *RACF Access Control Module Guide*. Db2 12 for z/OS. SC27-8858-02. Last accessed on 2018-3-1. IBM Corporation. 2017. URL: `https://www.ibm.com/support/knowledgecenter/SSEPEK/pdf/db2z_12_racfbook.pdf`.

[IBM17c] IBM. *Security Server RACF Security Administrator's Guide*. z/OS. SA23-2289-30. Version 2 Release 3. Last accessed on 2018-3-1. IBM Corporation. 2017. URL: `https://www-304.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3sa232289/$file/icha700_v2r3.pdf`.

[KKK06] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske. "Some Essentials of Graph Transformation." In: *Recent advances in formal languages and applications* 25 (2006), pp. 229–254.

[KMP02] Manuel Koch, Luigi V Mancini, and Francesco Parisi-Presicce. "A graph-based formalism for RBAC". In: *ACM Transactions on Information and System Security (TISSEC)* 5.3 (2002), pp. 332–365.

[KMP05] Manuel Koch, Luigi V Mancini, and Francesco Parisi-Presicce. "Graph-based specification of access control policies". In: *Journal of Computer and System Sciences* 71.1 (2005), pp. 1–33.

[SS94] Ravi S Sandhu and Pierangela Samarati. "Access control: principle and practice". In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

Appendix A

Mechanisms

Explicit privilege objects
Collections
Databases
Distinct types or JAR
Functions or procedures
Global variables
Packages
Plans
Routines
Schemas
Sequences
Systems
Tables and views
Usage
Use

Table A.1: Explicit privilege objects[IBM17a]

Table or view privilege	**SQL statements allowed for a table or view**
ALTER	ALTER TABLE, to change table definition
DELETE	DELETE, to delete rows
INDEX	CREATE INDEX, to create an index on the table
INSERT	INSERT, to insert rows
REFERENCES	ALTER or CREATE TABLE, to add or remove a referential constraint that refers to the named table or to do a list of columns in the table
SELECT	SELECT, to retrieve data
TRIGGER	CREATE TRIGGER, to define a trigger
UPDATE	UPDATE, to update all columns or a specific list of columns

Table A.2: Explicit table and view privileges[IBM17a]

Table name	**Records privilegs held for or authorization related to**
SYSIBM.SYSCOLAUTH	Updating columns
SYSIBM.SYSDBAUTH	Databases
SYSIBM.SYSPLANAUTH	Plans
SYSIBM.SYSPACKAUTH	Packages
SYSIBM.SYSRESAUTH	Buffer pools, storage groups, collections, table spaces, JARs, and distinct types
SYSIBM.SYSROUTINEAUTH	User-defined functions and stored procedures
SYSIBM.SYSSCHEMAAUTH	Schemas
SYSIBM.SYSTABAUTH	Tables and views
SYSIBM.SYSUSERAUTH	System authorities
SYSIBM.SYSSEQUENCEAUTH	Sequences
SYSIBM.SYSCONTEXT	Associating a role with a trusted context
SYSIBM.SYSCTXTTRUSTATTRS	Associating trust attributes with a trusted context
SYSIBM.SYSCONTEXTAUTHIDS	Associating users with a trusted context

Table A.3: Privileges information in Db2 catalog tables[IBM17a]

Profile name	**Profile type**	**COPY**	**COPY.PAPER**	**COPY.PAPER.TEST**
COPY.A	Discrete			
COPY.PAPER	Discrete		X	
COPY.PAPER.TEST	Discrete			X
COPY.PAPER.%	Generic			
COPY.PAPER.*	Generic			X
COPY.PAPER.**	Generic		X	X
COPY.PAPER%	Generic			
COPY.PAPER*	Generic		X	X
COPY.PAPE%	Generic		X	
COPY.*	Generic		X	X
COPY.**	Generic	X	X	X
COPY*.**	Generic			

Table A.4: Excerpt of a sample access to RACF resource profiles[IBM17c]

Db2 Object Type	RACF Member Class	RACF Grouping Class
Collections	MDSNCL	GDSNCL
Database	MDSNDB	GDSNDB
JAR	MDSNJR	GDSNJR
Packages	MDSNPK	GDSNPK
Plan	MDSNPN	GDSNPN
Schemas	MDSNSC	GDSNSC
Sequences	MDSNSQ	GDSNSQ
System	MDSNSM	GDSNSM
Table and views	MDSNTB	GDSNTB

Table A.5: Mapping between Db2 Object and RACF classes[IBM17b]

Appendix B

Access Control Migration

B.1 Derivation Examples

The following example set up of a database, its users, objects and privileges is used to illustrate the concepts and algorithms shown in chapter 4. The graphs are being derived with the rule set depicted in figure 4.4 and the graph is typed over the graph from figure 4.3.
The database DB has one table $T1$. That table is used by the users *Alice* and *Bob*. *Alice* and *Bob* are both members of the department $G1$, which has the privileges to SELECT and INSERT data on this table.
The database is administered by *Max*, who is assigned the DBADM authority. Additionally Max is a member of the department $G2$, which has the SELECT authority on table $T1$. The derivation to a graph will normally start with all administrative authorities, however for this example it will only start with DBADM. For a cleaner depiction of the graphs the names will be abbreviated to A, B and M, the privileges with S and I and the authority with ADM.

B.1.1 Graph Derivation from Db2

The catalog for the example database would look something like the table B.1. Not depicted are the authorities, however they are as mentioned in the introduction of the example. Since Db2 has no groups, every member of a department has to be granted the explicit privileges individually.

Table:

GRANTEE	OBJECTID	SELECTAUTH	INSERTAUTH
Alice	T1	Y	Y
Bob	T1	Y	Y
Max	T1	Y	N

Table B.1: Example catalog table for the Db2 object table

Applying the algorithm 1, as well as the not depicted one for the authorities to the start graph, containing only the authority vertex for the DBMADM authority, and the catalog table B.1 will result in the derivation sequence *(AddUser(A),AddObject(Table*

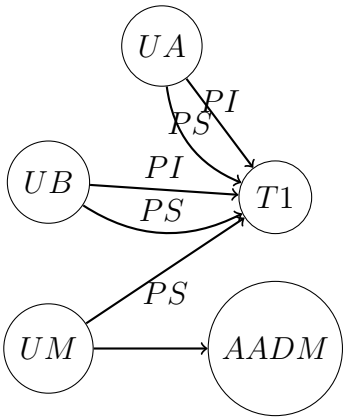


Figure B.1: Resulting graph derived from the example Db2 privileges

T1), AddUserPrivilege(A, Table T1, S), AddUser(B),...,AssignAuthority(M,DBADM). The resulting graph is depicted in figure B.1.

Applying the algorithm 2 to either figure B.1 or figure B.4 will result in the same set of explicit SQL-GRANTS, since the algorithm is defined on path and not simply on edges. Therefore it looks at the effective explicit privileges a user has. The resulting SQL-GRANTS can be seen in figure B.2. After executing these SQL statements the resulting catalog will look like the one shown in table B.1 in the beginning.

```
/*Explicit privileges based on paths*/
GRANT SELECT ON TABLE T1 TO ALICE;
GRANT INSERT ON TABLE T1 TO ALICE;
GRANT SELECT ON TABLE T1 TO BOB;
GRANT INSERT ON TABLE T1 TO BOB;
GRANT SELECT ON TABLE T1 TO MAX;
/*Administrative authorities*/
GRANT DBADM ON DB TO MAX;
/*Ownership is not being migrated*/
```

Figure B.2: Resulting SQL-Statements of Graph to Db2 Derivation

B.1.2 Graph Derivation from RACF

RACF has a group concepts which allows Max to grant privileges to the departments instead of the members of a department. While RACF stores its profile in a database, this database can not be directly queried like a Db2 database, but certain tools can extract files containing RACF profiles, which would look like figure B.3. Applying the algorithm 3 to the example scenario depicted in figure B.3 will result in the graph shown in figure B.4.

The algorithms 4 and 5 result in different outputs, depending on whether if it was applied to the graph of figure B.1 or figure B.4. The resulting RACF commands can

```
/*RACF resource profiles*/
CLASS     NAME
-----     ----
MDSNTB    DB.T1.SELECT

USER      ACCESS
----      ------
G1        READ
G2        READ
----------------------
CLASS     NAME
-----     ----
MDSNTB    DB.T1.INSERT

USER      ACCESS
----      ------
G1        READ
```

```
/*RACF group profiles*/
GROUP          GROUP
-----          -----
G1             G2

USER           USER
-----          -----
ALICE          MAX
BOB
```

Figure B.3: Example RACF profiles

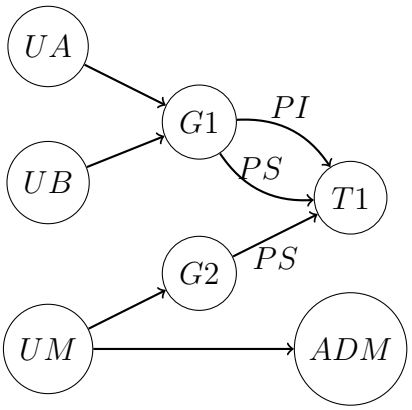


Figure B.4: Resulting graph derived from the example RACF privileges

be seen in figure B.5, the left side shows the commands derived from the Db2 graph and the right side the ones from the RACF graph.

```
/*Graph derived from Db2,
  RACF resource profiles*/
RDEFINE MDSNTB DB.T1.SELECT;
RDEFINE MDSNTB DB.T1.INSERT;
/*Privileges*/
PERMIT DB.T1.SELECT CLASS(MDSNTB)
   ID(ALICE) ACCESS(READ);
PERMIT DB.T1.INSERT CLASS(MDSNTB)
   ID(ALICE) ACCESS(READ);
PERMIT DB.T1.SELECT CLASS(MDSNTB)
   ID(BOB) ACCESS(READ);
PERMIT DB.T1.INSERT CLASS(MDSNTB)
   ID(BOB) ACCESS(READ);
PERMIT DB.T1.SELECT CLASS(MDSNTB)
   ID(MAX) ACCESS(READ);
```

```
/*Graph derived from RACF,
  RACF resource profiles*/
RDEFINE MDSNTB DB.T1.SELECT;
RDEFINE MDSNTB DB.T1.INSERT;
/*RACF group profiles*/
CONNECT UA TO G1;
CONNECT UB TO G1;
CONNECT UM TO G2;
/*Privileges*/
PERMIT DB.T1.SELECT CLASS(MDSNTB)
   ID(G1) ACCESS(READ);
PERMIT DB.T1.INSERT CLASS(MDSNTB)
   ID(G1) ACCESS(READ);
PERMIT DB.T1.SELECT CLASS(MDSNTB)
   ID(G2) ACCESS(READ);
```

Figure B.5: Resulting RACF commands of Graph to RACF Derivation

Appendix C

Implementation

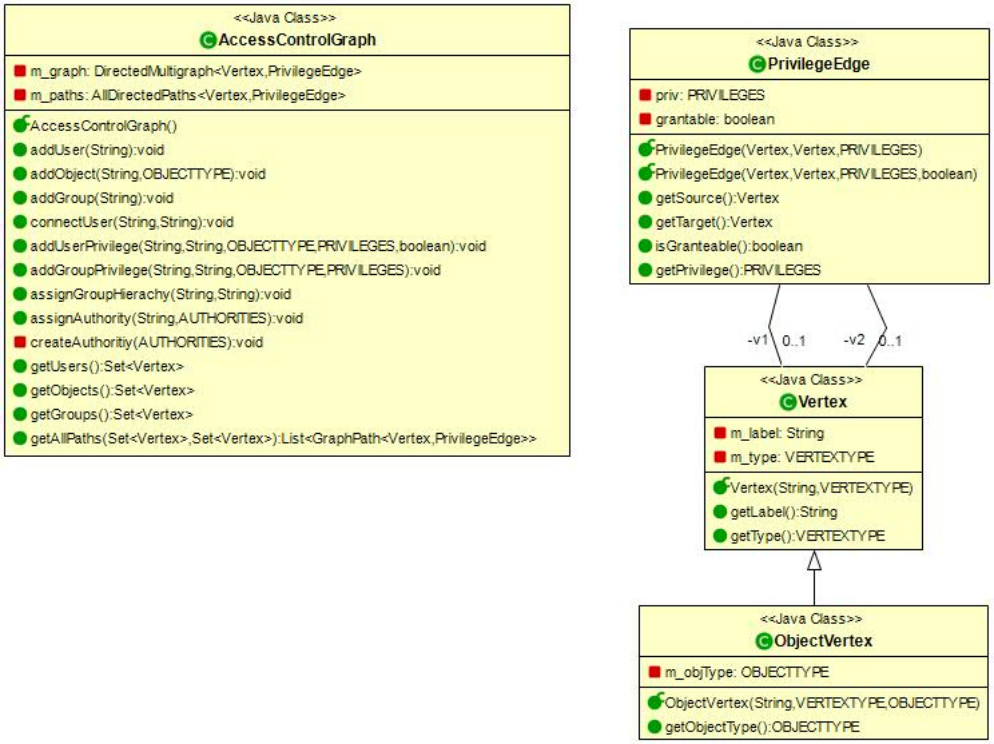


Figure C.1: UML diagram for AccessControlGraph and related classes

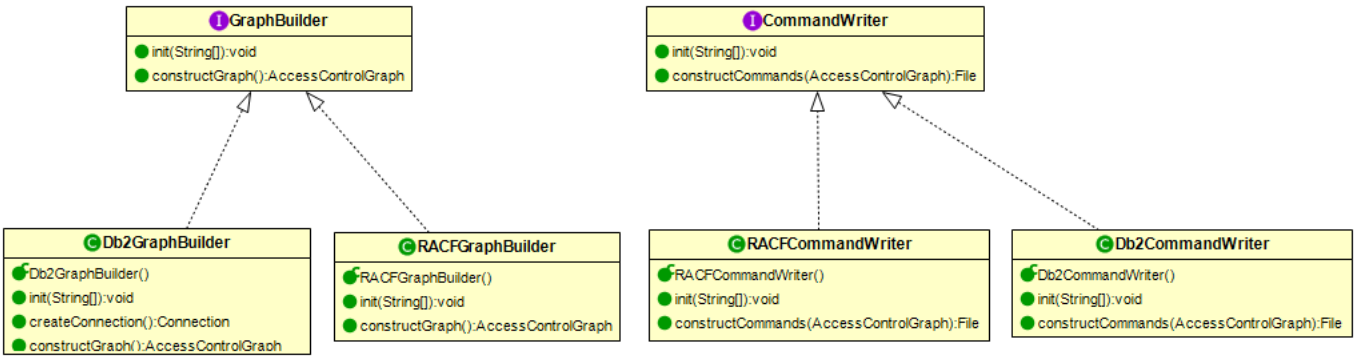


Figure C.2: UML diagram for the interfaces and the classes that implement them