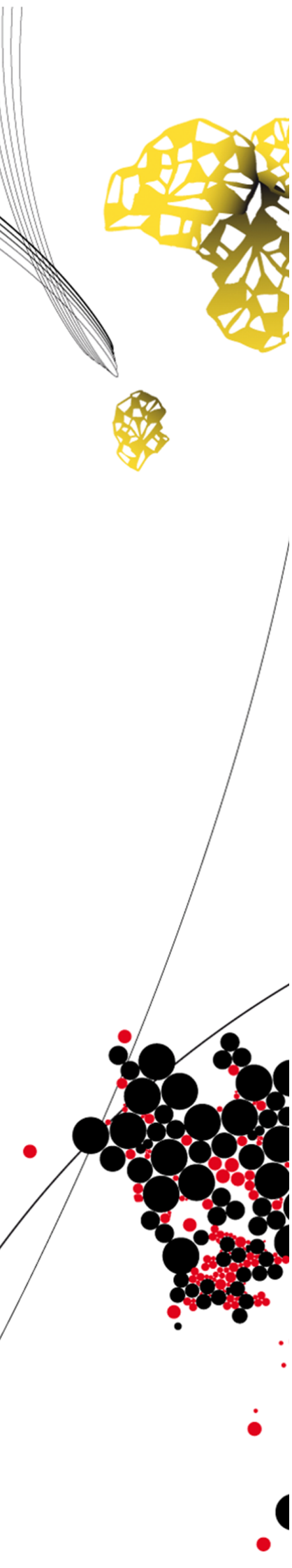


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Incremental Verification of Physical Access Control Systems

J.H. van der Laan

Master Thesis
January 2021

Supervisors:
prof. dr. M. Huisman
dr. W. Kuijper
Advisor:
dr. R.E. Monti
Industrial Advisor:
H.A.J. Berntsen MSc.

Formal Methods and Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

Chapter 1

Introduction

From pharmaceutical employees stealing drugs, heroin disappearing from police evidence lockers prior to trials and disgruntled contractors releasing personal credit card information [1]: small security holes can have large consequences for organizations. One of the central pillars of organizational security is physical access control. Physical access control protects people and resources against unauthorized access and the disastrous consequences which could stem from it.

The goal of physical access control is to manage who can access a location and when. When we talk about physical access control systems, we usually denote electronic access control systems in which people use an identifier, such as an ID badge, to gain access to locations protected by measures, such as electronic door locks. The person trying to get access could be an employee, a contractor or a visitor. The location they are trying to access could be a site, a building or a single room.

Managing physical access control systems is a complex task. As organizations change and so does the world around them, their security systems must adapt to their changing needs. While establishing an initial access control policy is relatively easy, evolving it is much harder. Not only can these policies be very intricate and convoluted, underlying policy invariants are also often implicit or written down outside of the system. These characteristics make it difficult to foresee and assess the influence of a change in the system or modification of the policy. This is dangerous since a misconfigured access control system poses a security risk and can render employees unable to perform their daily tasks.

Nedap has also identified the problems above and is actively working towards ways

[1] examples based on anonymized incidents presented in `https://www.securityinfowatch.com/access-identity/article/12293604/fixing-the-gaps-in-your-pacs` (accessed on 22-1-2021)

to help security officers tackle the complexity. They have a department that builds software and hardware for physical access control systems [2]. At the time of writing, they have set up an R&D team to develop the next generation of physical access control software.

Among other things, they have developed a robust access control authorization model formalism called *bi-sorted role-based access control* (RBÄC) [1] to alleviate some of the burdens. Authorization models are consulted to determine whether somebody should be granted or denied access when requested. RBÄC is a fragment of *role-based access control* (RBAC) [2], implying that RBÄC might be directly used with existing RBAC implementations [1]. The idea behind RBAC and RBÄC is that a security officer can govern what people, also called users, can access by assigning roles to users (e.g. receptionist, security guard, contractor or a visitor), and controlling what permissions users with these roles should have (e.g. what locations they should be able to access).

Another way of tackling the complexity is through formal verification. Access control policy verifiers can be used to check that a policy does not contain security or configuration mistakes. In our research, we would like to help Nedap facilitate physical access control policy evolution through formal verification by checking policy invariants on authorization models, also commonly known as *authorization constraints*.

Problem Statement

Unfortunately, most existing access control policy verifiers are not designed to analyze evolving policies. They analyze policies in their entirety (e.g., all verifiers shown in the recent survey paper [3]). In practice, this means that the entire policy and all policy invariants have to be reverified from scratch after each small change. This is often unnecessary and highly inefficient when dealing with evolving policies.

We believe a better approach would be to verify policies *incrementally*, i.e. to reuse intermediary computations and results from previous verifications. This would make it is possible to limit the set of computations to redo; constraints to recheck; and/or the set of model entities to reconsider upon a change. We argue that therefore incremental verification would be a much more efficient way to verify evolving policies. Thus in this research, we will explore the following problem:

[2] `https://www.nedapsecurity.com/nl/`

Problem Statement: *How can we incrementally verify authorization constraints on realistic RBÄC-inspired evolving physical access control systems?*

To the best of our knowledge, incremental verification has only been sparingly explored in the domain of access control verification [4] [5] [6] and never in the domain of physical access control verification. However, it has been successfully applied in many other application areas such as in: product line engineering configuration verification [7] [8], OCL checking [9] [10], validation of XML documents against DTDs [11] [12] and program analysis [13].

Concretely, we will thus help Nedap facilitate physical access control policy evolution by defining an RBÄC-inspired authorization model and showing how realistic evolving physical access control systems captured in that model can be verified. Specifically, we are going to focus on checking changes will not violate authorization constraints. This will help security officers to foresee and assess if a change introduces security/configuration mistakes (through pre-defined authorization constraints) and if policy invariants which would previously be written down outside are still respected (through user-defined authorization constraints). Ideally, security officers would be able to continuously verify the authorization model and get quick feedback after each change.

Research Questions

The main problem can be subdivided into three separate yet connected problems. These smaller problems naturally lead us to our research questions.

Incremental Authorization Constraint Verification

The first problem is that the authorization constraints should be incrementally verified. Incremental graph pattern matching has achieved notable success to incrementalize verification and other types of analysis in multiple domains [10] [13] [14]. We will explore if this technique is also suitable to verify authorization constraints. We will do this with the help of VIATRA, an open-source incremental model query and transformation framework based on graph patterns [15]. Thus, our first research question will be:

RQ1: *Can we verify authorization constraints incrementally with graph patterns?*

We do not know of any prior research in which incremental graph pattern match-

ing has been used to verify authorization constraints. To the best of our knowledge, it has only been used to operationalize access control policies [16] [17].

We will explore if common authorization constraint types from the literature and domain-specific authorization constraint types (established in collaboration with Nedap) we expect security officers want to verify can be incrementally verified with the help of this framework. When an authorization constraint is violated, we want to be able to find all reasons why that constraint is violated. When the authorization model is changed, we want to be able to detect which authorization constraints became violated or resolved due to the change.

Time-dependent Accessibility

The second problem is that the solution should be able to analyze realistic physical access control policies. However, RBÄC does not support time-dependent accessibility. This is a requirement for realistic physical access control policies. Very often, employees with roles such as night-time security guards and cleaners have different permissions during their shifts compared to outside their shifts. As our goal is to analyze realistic policies, we will try to propose an RBÄC-inspired authorization model such that authorization can be granted or withheld based on time. Since we explore if we can verify policies incrementally, we should design this formalism with this in mind. Thus, our second research question will be:

RQ2: *How can we design an RBÄC-inspired authorization model with time-dependent authorizations in a way that is suitable for incremental verification?*

We will explore how this can be done while trying to keep the desired properties and the spirit of RBÄC.

Context-dependent Authorization Constraints

The last problem is that our solution should be able to verify the types of authorization constraints security officers want to verify. Next to authorization constraints which only depend on the authorization model, security officers also want to specify and verify authorization constraints on the authorization model which do not only depend on information contained in the authorization model. This is because in physical access control, security and configuration mistakes on the level of the authorization model can be inherently dependent on the context the model is deployed into.

Two key context-dependent authorization constraints security officers often want to check in physical access control are:

- users can never get trapped, i.e. users should always have a set of permissions which allows them to leave the collection of areas under access control.
- users can always enter the security zones he/she has the permission to access, i.e. users should always be able to invoke all granted permissions.

These authorization constraints do not only depend on the authorization model. Instead, they depend on the interplay between the authorization model, the access control system topology and the authentication model. From the access control system topology, we know how groups of locations delimited by access control measures, which we will call *security zones*, are interconnected and connected with the outside. From the authentication model, we know the *obligations* (authentication requirements) (none, access code, id badge, iris scan, etc.) to access a security zone depending on the time.

To the best of our knowledge, it has not yet been explored how an authentication model and the building topology can be linked with the authorization model in such a way that security officers may use this information to prove properties about the authorization model. Our third research question will thus be:

RQ3: *How do we account for authorization constraints dependent on the access control system topology and the authentication model in a way that is suitable for incremental verification?*

We will focus on verifying the two aforementioned key invariants and contextual factors. Although we could take more into account, we believe these two invariants and factors provide an interesting jumping-off point to explore how we could deal with context-dependent authorization constraints.

Performance Evaluation

To know if the presented solutions are a step in the right direction, we need to know if our approach could be efficient/scalable enough for real systems. This leads us to the fourth and final research question:

RQ4: *How efficient and scalable is our approach?*

We will explore this question by implementing our ideas into a prototype and mea-

suring the performance of this prototype when analyzing an actual physical access control system of an anonymous company. This data has been graciously provided to us by Nedap. We have released an anonymized version of the security policy alongside this research.

Thesis Structure

This thesis is structured as follows:

- In Chapter 2, Background, we present the access control domain, access control analysis, incremental verification and VIATRA.
- In Chapter 3, Related Work, we show the current state of the art of access control verification and physical access control modelling.
- In Chapter 4, GR-RBAC, we present and define our RBÄC-inspired authorization model which allows for time-dependent accessibility.
- In Chapter 5, Site Access Control System Model, we present and define an access control system topology model, an authentication model and the Site Access Control System (SACS) model which encapsulates all previously introduced models.
- In Chapter 6, Authorization Constraints, we present and define the authorization constraints (sub)types we want to verify incrementally.
- In Chapter 7, Implementation, we show how we constructed a prototype incremental policy verifier with VIATRA based on our previously defined models and authorization constraints (sub)types.
- In Chapter 8, Evaluation, we evaluate our prototype based on an actual physical access control system.
- In Chapter 9, Final Thoughts, we conclude our research, present our secondary contributions, discuss the limits of our research and point to possible future work.

Chapter 2

Background

2.1 Access Control

Access Control is a security mechanism which regulates who can access what in a cyber(-physical) system. Depending on the system, accessing may mean consuming, entering, using, dowloading, deleting, etc. It may mean anything from entering a room to reading a document. It depends on the system under access control. The goal of access control is to reduce the risk of unauthorized access, thereby enhancing the integrity and confidentially of the system and its resources. In the access control domain, the entities which can perform actions on the system are called *users* or *subjects*. The entities representing resources which might require permission to be accessed are called *objects*.

Access Control may be implemented on different levels and different types of systems, such as on databases, operating systems but also buildings. Depending on the system, examples of users may be persons or non-person entities such as processes, servers or routers. Examples of objects could be areas, files or networks.

Inspired by the definitions presented in [18], we can say that conceptually there are three parts to access control:

1. *Identification*: The means a user uses to claim an identity, role or attribute. This can be done with ID badges, user names, process ids, MAC addresses or anything else which can uniquely identify users.
2. *Authentication*: The means used to prove the right to use an identity, take on a role or prove possession of one or more attributes. Authentication can be achieved through 3 different approaches: based on something the user knows (e.g. passwords, keys), based on what the user has (e.g. ID badges, RSA

tokens) or based on what the user is or does (e.g. fingerprints, iris scans).

3. *Authorization*: The means of expressing the access policy by explicitly granting or withholding permissions.

The result of the identification, authentication and authorization process to grant or withhold access when requested is called an *access decision* [18].

To get a better understanding of the above concepts, let us consider the access control system of a building. There:

- Identification could be done by handing out ID badges to employees,
- Authentication could be done through badge scanners,
- Authorization to enter could be done based on a time-table which indicates when users are working and thus should be able to enter the building.

During our research, we will focus on the authorization and authentication aspect of access control. We will not take the identification process into account.

2.2 Types of Access Control

There are two main categories of access control authorization methods: *Discretionary Access Control* (DAC) and *Non-Discretionary Access Control* (NDAC) [19]. The experience of Nedaps security experts tells us that NDAC policies are more appropriate for the physical access control domain than DAC policies. Therefore, in this section, we will only briefly explain DAC and give a more in-depth introduction into NDAC policies.

Discretionary Access Control (DAC) gives users all permissions related to objects they have created or have previously been given access to [20]. This includes sharing permissions regarding these objects with other users. With DAC, invocation and sharing of permissions are up to a user's own discretion, hence the name.

Non-Discretionary Access Control (NDAC) can be seen as the inverse of Discretionary Access Control. With NDAC, a centralized authority is in charge of the access control system [19].

In contrast to DAC, depending on the form of NDAC, a user that has been granted access to an object may be limited in doing any of the following [21]:

1. passing the object to unauthorized users or objects,

2. granting permissions to access the object to other users,

3. changing one or more security attributes on users, objects or any other elements of the system,

4. choosing the security attributes to be associated with newly-created or modified objects,

5. changing the policy and it rules governing access control.

The two most popular types of NDAC are arguably *Role-Based Access Control* (RBAC) and *Attribute-Based Access Control* (ABAC).

2.2.1 RBAC

The idea behind Role-Based Access Control (RBAC) is that users are assigned roles which in turn are assigned permissions based on a policy. [2]. DAC can be emulated with RBAC [22].

Many extensions for RBAC have been defined, which add support for concepts such as spatial-and location-based information (GEO-RBAC [23]), time (TRBAC [24], GTRBAC [25]), both location and time (STRBAC [26], GSTRBAC [27]), indirection based on grouping permissions (T-RBAC[28], RBÄC [1]) and withholding previously granted permissions ($R^{\pm}$BÄC [1]).

2.2.2 RBÄC

RBÄC is an RBAC extension developed by security experts of Nedap. In RBÄC, permissions are assigned to *demarcations* which are in turn assigned to roles instead of directly assigning permissions to roles. RBÄC subsumes RBAC.

This extension was developed as a reponse to the observation that plain RBAC "blends together subject management aspects and permission management aspects into a single object of indirection: a role"[1]. Demarcations decouple the tasks of maintaining abstractions over the set of users (assignment of users into roles), maintaining abstractions over the set of permissions (assignment of permissions into demarcations), and maintaining abstract access control policy (granting roles access to demarcations) [1]. This decoupling facilitates policy evolution.

2.2.3 ABAC

Attribute-Based Access Control (ABAC) [29], sometimes also referred to as policy-based access control (PBAC)[18] and rule-based access control (REBAC, e.g. in [20]), gives permissions to users based on policy rules evaluated on attributes. The rules can be defined on any type of attributes (user attributes, object attributes, system attributes, etc.). ABAC is limited only by the computational language and the richness of the available attributes [30]. RBAC can be emulated with ABAC. Work has been done on combining RBAC and ABAC [31].

2.3 Authorization Constraints

As briefly touched upon in the introduction, authorization constrains are policy invariants defined on the authorization model. With basic RBAC, four common types of authorization constraints are:

1. *Cardinality constraints*
2. *Separation of duties (SoD) constraints*
3. *Binding of duties (SoD) constraints*
4. *Prerequisite constraints*

With cardinality constraints, we constrain the number of entity occurrences that are associated in a relationship. In the context of RBAC, this means we can put a bound on the number of roles assigned to a user of the number of permissions assigned to a role. Then, a user can only have a limited amount of roles at the same time.

With SoD constraints, we define mutual exclusion relations between entities. Two mutual exclusive entities cannot be in relation with the same entity. For example, we can define two mutually exclusive roles and enforce that a user should never have them assigned at the same time.

With BoD constraints, we define mutual dependency relations between entities. Two mutual dependent entities must always be in relation with the same entity. For example, we can define two mutually dependent roles and enforce that a user should always have them assigned at the same time.

With prerequisite constraints, we require that certain relationships between entities may only exist given that other relationships between entities exist. This constraint can, for example, enforce that a user can only be assigned a certain role if and only

if he/she already has another role.

There are many more types of constraints and RBAC extensions often introduce new types of constraints. A comprehensive review can be found in [32].

2.4 Access Control Analysis

Access control analysis is crucial to support policy creation and evolution. It is commonly carried out for two different purposes: verifying that the policy conforms to a set of quality requirements and supporting the design and organization of a set of policies [3].

In the context of RBÄC, assessing the quality of your access control policy entails checking that the policy is:

- *consistent*: the authorization model contains no contradictions,
- *minimal*: there is no unnecessary redundancy,
- *revelant*: there are no irrelevant assignments,
- *complete*: for each access request, the model contains the necessary information to either grant or withhold the permission,
- *correct*: all authorization constraints hold.

We call access control verification the subset of access control analysis which only deals with checking consistency, completeness and correctness.

In RBÄC, consistency and completeness are enforced by design. It is not possible to specify contradictions nor leave parts of the system under specified. This is because permissions are withheld unless specifically granted, and only grants can be specified.

Depending on the context, minimality is not always a desirable property. Redundancy can be good in the face of change. Arguably, minimality might even be a too strong requirement in general

The design and organization process can be supported through *change impact analysis*, *similarity analysis* and *policy set structuring* [3]. A change impact analysis assesses and evaluates a change on a single policy or a collection of policies. This analysis identifies the potential consequences and helps to estimate the associated

risks. Similarity analysis performs comparisons among two or more policies. Specifically, it checks for different relationships among policies such as equivalences, possible refinements, redundancies and conflicts [33]. This is an important analysis for policy integration (merging policies). Policy set structuring analyses the structure of a single or a collection of policies. It helps to ensure the optimal policy structure. A well-known example of this kind of access control analysis is role mining [34].

2.5 Incremental Verification

There are many incremental computation approaches one could use for incremental verification. Examples include using graph pattern matching (e.g. [15] and [35]) and verification trees (e.g. [8] and [9]). However, no matter the approach all incremental checkers try to accomplish the same goal: limiting the set of rules to recheck (*rule reduction*) and/or the model elements to consider (*scope reduction*) after each model change [36].

Next to having the same goal, incremental checkers also have shared feature sets [36]. To recheck rules, they must have access to the model, and it must be able to process model changes. Moreover, it must have some method to perform rule and/or scope reduction. To speed up the reduction, many incremental checkers use a form of caching (often for some form of memoization). If the approach uses a cache, it requires a method to manage it. Usually, when a cache is introduced, it is accompanied by some kind of pruning method.

The downside of caching is that it may introduce a significant memory overhead. This introduces a fundamental balancing act between efficiency (more caching) and scalability (less caching). There is no silver bullet. The many different approaches to incremental computation and verification all stem from an effort to optimize efficiency, scalability and their balance for various application domains and use-cases. The different methods have different strengths and weaknesses depending on the shape of the data and the nature of the checks.

2.6 Viatra

For our research, we are going to use the VIATRA[1] model query and transformation framework [15]. This framework supports incremental consistency checking through incremental graph queries. In this section, we will briefly go over the most important aspects of this framework which are necessary to understand the work presented in this thesis. For more information about VIATRA, we refer the reader to: `https://www.eclipse.org/viatra/documentation/index.html`.

VIATRA is an open-source model query and transformation framework which supports event-driven, reactive transformations and incremental graph queries [15]. The framework supports Ecore and UML models made with the *Eclipse Modelling Framework* (EMF). Graph patterns can be defined with a declarative DSL called the *Viatra Query Language* (VQL). The queries can be evaluated incrementally over models with millions of elements. [37]–[39].

2.6.1 Model Support

VIATRA supports Ecore and UML models made with the Eclipse Modeling Framework (EMF) [40]. EMF can be considered a de-facto standard for modeling. Ecore and UML allow the defintion of *meta-models*, namely models which described the structure of other models. In our research, we will implement the meta-model of our formalism as an Ecore model. The core of the Ecore model itself can expressed using four types of elements: `EClass`, `EAttribute`, `EDataType` and `EReference`. To create a new meta-model, these elements can be used as follows:

- Classes can be modeled with `EClass` entities. Classes can contain a number of attributes and references. Multiple inheritance is supported between classes.
- Class attributes can be modeled with `EAttribute` entities.
- The attribute datatypes can be express with `EDataType` entities. They represent atomic values (i.e. any element whose internal structure is not modeled).
- The association between model classes can be captured with `EReference` instances. Bidirectional references are expressed with two opposite references.

An overview of the basic Ecore elements is shown in Figure 2.1. An example RBAC metamodel and model can be found in Figure 2.2. Ecore contains many additional elements helpful for the organization of the models.

[1] https://www.eclipse.org/viatra/

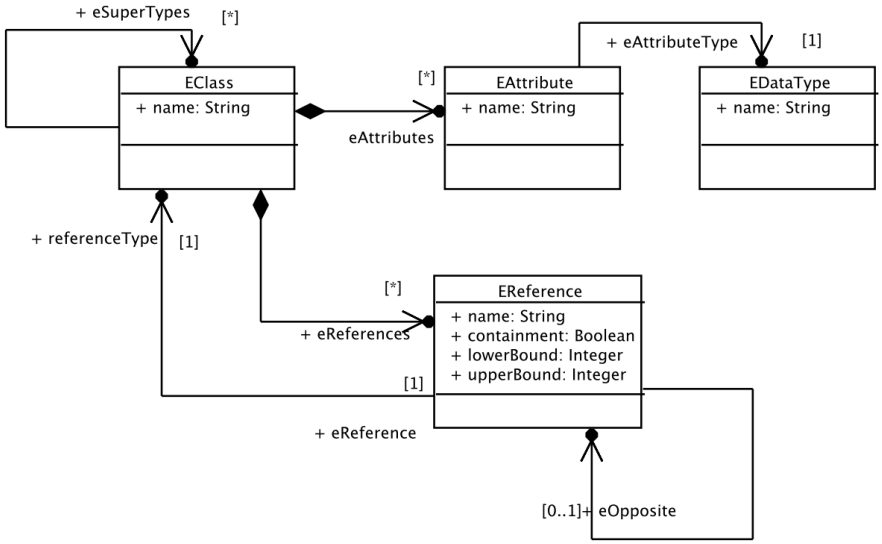
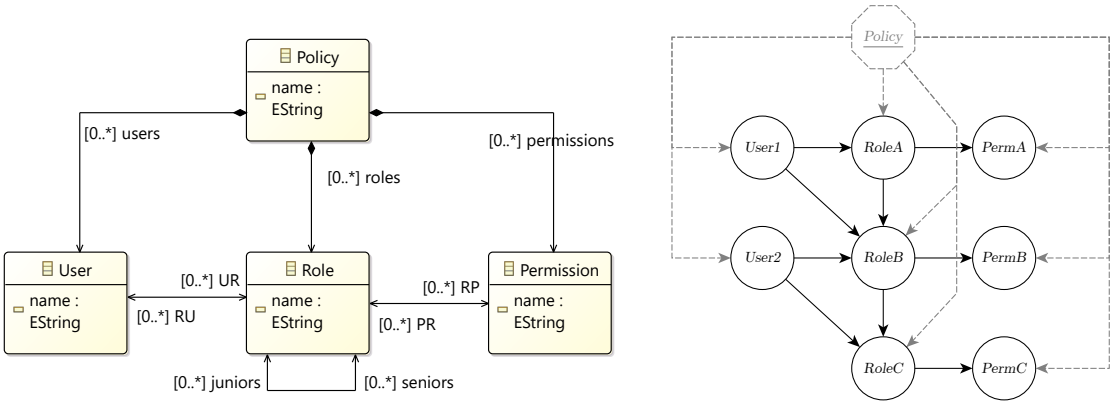


Figure 2.1: Core of Ecore metamodel [40]



(a) RBAC metamodel (b) RBAC model

Figure 2.2: RBAC metamodel and model

2.6.2 Query Language

Model queries in VIATRA can be defined as graph patterns using the Viatra Query Language (VQL) [40]. Graph patterns are expressed declaratively as a set of constraints. A tuple of parameter variables that fulfill the graph pattern is called a *match*. The set of all matches of a graph pattern is called the *match set*. Pattern variables that are not pattern parameters are called *local variables*.

VQL is an expressive language, capable of expressing constraints like:

- Path Expressions: a specific reference, an attribute, or a path of references must exist between two variables.

```
pattern UP(user: User, perm: Permission) { // UP = UR;RP
    User.UR(user, role); // role (local var.) should be linked to
    user with relation UR, i.e. role should be assigned to user.
    Role.RP(role, perm); // perm should be assigned to role.
}
```

  The match set of this pattern for Figure 2.2 is $\{(User1, PermA), (User1, PermB), (User2, PermB), (User2, PermC)\}$.

- Transitive Closure: compute the (reflexive) transitive closure of binary patterns in a pattern call. $+$ indicates the transitive closure, $*$ the reflexive transitive closure.

```
pattern AccessRelation(user: User, perm: Permission) {
    User.UR(user, role);
    Role.juniors*(role, roleP); // reflexive transitive closure
    Role.RP(roleP, perm);
}
```

  The match set of this pattern for Figure 2.2 is $\{(User1, PermA), (User1, PermB), (User1, PermC), (User2, PermB), (User2, PermC)\}$.

- Attribute Equality: an attribute of a variable (class instance) must be equal to a given variable or value.

```
pattern SoDRoleBAndRoleC(user: User) {
    Role.name(roleB, "RoleB"); // the value of the name attribute
    of roleB should be "RoleB", i.e. roleB should be named "RoleB"
    Role.name(roleC, "RoleC"); // roleC should be named "RoleC"
    Role.RU(roleB, userB); // roleB should be assigned to userB
    Role.RU(roleC, userC); // roleC should be assigned to userC
    user == userB; // user should be equal to userB
    user == userC; // user should be equal to userC
```

```
}
```

The match set of this pattern for Figure 2.2 is $\{(User2)\}$ (violations of the constraints are matched).

- Pattern Composition: a given set of variables must or must not match to (another) pattern.

```
pattern PrerequisitePermAImpliesPermB(user: User) {
    Permission.name(pA, "PermA"); // pA should be named "PermA"
    find AccessRelation(user, pA); // the tuple (user,pA) should
    be in the match set of AccessRelation
    Permission.name(pB, "PermB"); // pB should be named "PermB"
    neg find AccessRelation(user, pB); // the tuple (user,pA)
    should NOT be in the match set of AccessRelation
}
```

The match set of this pattern for Figure 2.2 is $\{\}$.

- Check Expression: an arbitrary expression containing attributes must be evaluated true

```
pattern CardinalityMaxOneRoleB() {
    Role.name(role, "RoleB"); // role should be named "RoleB"
    n == count Role.RU(role, _u);  // n should be equal to the
    number of matches of Role.RU(role,_u), i.e. the number of
    users assigned RoleB.
    check(n >= 2); // n should be >= 2
}
```

The match set of this pattern for Figure 2.2 is $\{()\}$.

Next to the constraints shown above, the language has a few additional useful constructs such the `eval()` expression which can evaluate any arbitrary pure (no side-effects) Java expressions and aggregators such as `min/max/sum/count` which calculate the minimum/maximum/sum/number of matches of a path expression or pattern has. The language can be extended with custom aggregators. In our research, we have defined a custom aggregator which given a match set converts it to a Java set.

For a complete overview of the language, we refer the reader to [41]. However, please note that the syntax has changed a bit since the publication of the paper.

2.6.3 Query Evaluation

The Viatra framework currently supports two graph pattern evaluation techniques: *local search* and *Rete*.

With local search, the matching process is started from a single node and extended step-by-step with the neighboring nodes and edges following a generated search plan. Local search is non-incremental: upon a model change the matching process has to be redone from scratch.

In contrast, Rete is incremental. With Rete, the (sub)pattern match sets are cached. These caches are organized in a graph structure called a Rete Network and incrementally updated upon model changes. When a match is added or removed from a (sub)pattern's match set because of model changes, the patterns depending on this (sub)pattern are partially re-evaluated based on the added/removed matches.

Local search based pattern matching has a much smaller memory footprint and shorter initialization phase compared to incremental approaches. However, incremental approaches, such as Rete, can provide an order of magnitude faster re-evaluation time [39]. This comes back to the fundemental balancing act between efficiency and scalability in incremental verification introduced in 2.4. As in our research we focus on incremental verification, we use Rete instead of local search to and evaluate graph queries incrementally.

More about the implementation of local search and Rete in VIATRA can respectively be found in [42] and [43].

Chapter 3

Related Work

In this chapter, we will present work related to our research. Literature regarding incremental verification of access control policies is scarce. Thus, in this section, we will mostly show works relevant to our research regarding modelling and non-incremental verification of (T)RBAC and physical access control policies.

3.1 (T)RBAC Modelling & Verification

Much research has been done regarding the analysis of RBAC policies. Many formalisms have been applied to model and verify RBAC policies such as first-order logic [44], modal logic [45], situation calculus [46], Petri-nets [47] and Kripke structures [48]. Extensive overviews and applications are presented in [3] and [49].

The TRBAC extension has also been the subject of much research, although not to the same degree as plain RBAC. Attempts to model and verify TRBAC policies have used formalisms such as first-order logic [50], Petri-nets [49] and timed-automata [51].

Although not yet popular, speeding up policy verification through incrementalization has been somewhat explored. An incremental algorithm to check dynamic separation of duty constraints with the help of information flow graphs was introduced in [4]. Additionally, incremental algorithms to verify administrative access control safety properties (i.e. ensuring there are no sequences of administrative actions which can result in policies by which can compromise some security goals) were presented in [5] and [6].

Despite the fact that these works share the same insight that policy analysis can be

sped up by taking an incremental approach, their goals and approaches are quite different from ours. We hope to support a broader scope of constraints than [4]. On the flip side, we do not aim to support most of the types of constraints presented in [5] and [6] (e.g. is it possible for an administrator to assign a specific role to a specific user?). These works also present algorithms tailored to verify specific types of constraints while we propose to use a more general technique to verify a broader scope of constraints.

Works which are arguably more similar to our approach are the many other attempts at formalizing RBAC constraints with the help of another declarative model query language called OCL (Object-Constraint Language). As with VQL, OCL can put constraints on Ecore and UML models. However, OCL does not support incremental evaluation out of the box. Thus, the queries and the models in these works have not been designed with incremental verification in mind. Research has however been carried out on how to incrementally evaluate OCL constraints, including an attempt to translate a subset of OCL to VQL [10] (back then called EMF-IncQuery).

Two notable attempts of formalizing access control constraints with OCL are [32] (GemRBAC), and [52]. Both works are not only capable of analyzing RBAC models/-constraints but also extensions such as TRBAC. The main difference between them is that [32] focused on supporting many types of constraints while [52] focused on creating an access control model which supports multiple NDAC formalisms (DAC, RBAC and ABAC). Performance was not addressed in [32]. In [52], it was stated that performance was a disadvantage of their approach. However, improvements have been proposed by the authors, which might increase performance [52].

Other works where OCL was used for the formalization of RBAC constraints include [53]–[60]. The EMF in general has also been used as the basis to support other policy analysis activities such as generating migration guides [61] and facilitating policy mutation for mutation testing [62]. VIATRA has also been used before in the context of access control, however not to analyze policies but instead to enforce rule-based access control policies at runtime [16].

3.2 Physical Access Control Modelling & Verification

In contrast to (T)RBAC, not a lot of work has been done regarding modelling and verifying physical access control policies. To the best of our knowledge, speeding

up policy verification through incrementalization has not been explored in the domain of physical access control.

Various RBAC extensions have been proposed to deal with geo-spatial contexts such as *GEO-RBAC*[23], *STRBAC* [26], *GSTRBAC* [27] and the previously presented GemRBAC. These models allow the enabling/disabling of roles, the specification of constraints and more (depending on the formalism) based on the physical location of various entities. These models will serve as an inspiration during our research but will not be fully incorporated into our work. This is because, during our research, we will not take into account policies where the physical location of entities influences the policy behaviour. For example, we will not deal with policies which enable/disable roles based on a user's physical location in a building.

To the best of our knowledge, the only tool which is similar to us in terms of goals is the physical access control management tool presented in [63]. This tool allowed security experts to explore, visualize and analyze access control policies by utilizing a building information model (BIM). BIMs are essentially digital twins of a building. They can contain information regarding objects and processes within a building [63].

The access control model behind the tool in [63] is a combination of ABAC and RBAC (based on [31]). The policies were modelled with the help of the EMF. The tool supports constraints on entity assignment (e.g. separation of duty). The authors did not mention how they checked the constraints nor if any rule/scope reductions were made.

The authors of [63] also noticed the requirement that users should be able to reach all rooms they have the permission to access. They proposed to solve this problem by enforcing reachability on the level of roles: whenever a role is given access to a room, the role should always have a path to this room. Thus, whenever the security officer assigns a role to a user; the security officer knows the user can reach all rooms the user has the permission to access.

A graph-based formal representation of BIMs specifically for physical access control applications is presented in [64], by (predominantly) the same authors of [63]. The model is based on hierarchical graphs and partitions buildings into storeys and spaces. The nodes and edges represent the spaces and the connections between those spaces. The nodes and edges can be associated with additional attributes to capture relevant security aspects such as security clearances, security requirements and space/door types. Next to the model, the authors present multiple utility

algorithms: a basic pathfinding algorithm, a minimum-security pathfinding algorithm and a reachability algorithm which checks if a given user can reach a specific room from another room.

Chapter 4

Gregorian RBAC

Time-dependent accessibility is a requirement for realistic systems. Since we want to analyze RBÄC-inspired authorization models and RBÄC does not support this functionality, we will extend this formalism with time-dependent accessibility. We will call this extension *Gregorian RBAC* (GR-RBAC) as how it deals with time is heavily tied to the Gregorian calendar.

In this chapter, we will introduce GR-RBAC. We will do this in five steps. First, we will present the formalism we inspire ourselves upon: RBÄC. Then, we will show some extensions which add time-dependent accessibility to RBAC and pick a promising candidate to rebase onto RBÄC. Next, we will discuss what roadblocks exist to rebase this extension onto RBÄC (for the domain of physical access control) and how they might be overcome. Lastly, we will show how we combined RBÄC and the chosen extension into GR-RBAC and provide formal definitions for this new formalism.

We will use the formal definitions in the next chapters when we introduce a formalism which ties the authentication model and the building topology together in an access control system model and when we formalize the authorization constraint types (on the GR-RBAC-based authorization model) security officers want to verify. Additionally, these definitions but also the definitions we presented in future chapters will form the basis of the Ecore metamodel and the VQL graph patterns of our prototypical implementation.

4.1 RBÄC

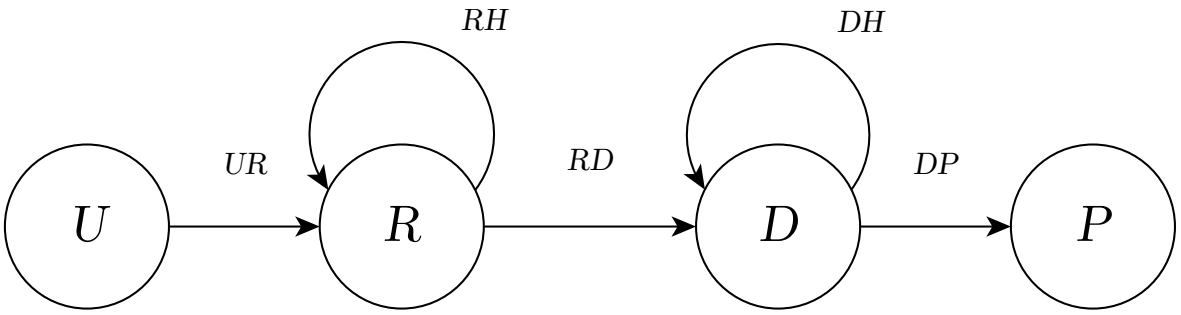


Figure 4.1: RBÄC Model Overview

An RBÄC authorization model consists of four components: a set of users U, a set of roles R, a set of demarcations D and a set of permissions P. No distinction is made between having a role and activating it (i.e. the assumption is made that a subject will always activate its roles). Thus, RBÄC does not deal with sessions (in contrast to plain RBAC).

In the context of physical access control, users are generally the employees of the organization. They are often also called subjects. Roles can be assigned to users and generally correspond to positions in the organization. They are thus meaningful in the context of the organization. In essence, demarcations are groups of permissions. Roles are associated with a set of demarcations, i.e. the groups of permissions required to carry out the role effectively. Permissions give access to an object. In physical access control, permissions give access to areas or allow users to open specific doors. On both roles and demarcations, a hierarchy is defined. Roles higher in the hierarchy (*senior roles*) inherit the demarcation assigned to roles lower in the hierarchy (*junior roles*). Demarcations higher in the hierarchy inherit the permissions assigned to demarcations lower in the hierarchy.

In RBÄC, the assignment of users to roles, roles to demarcations and demarcations to permissions is captured by the user-role assignment relation UR, role-demarcation assignment (grant) relation RD and the demarcation-permission assignment relation DP (see Figure 4.1). Taking into account the hierarchies, we can derive the so-called *access relation* UP between users and permissions which contains who has which permission.

Practitioners have different views on what demarcations should represent. One side argues that demarcations should represent processes. Then, assigning permissions to a demarcation equals to stating what permissions are required to carry out

that process. Furthermore, granting demarcations to a role equals to stating which processes a role is allowed to carry out. The other side argues that demarcations should be an abstract concept and what a demarcation represents should be left as an implementation issue. Then, the meaning behind assigning permissions to a demarcation and granting demarcations to a role depends on the meaning given to demarcations by the implementation. In this research, we will not require demarcations to represent any specific concept at all and thus the extension will be useful for all practitioners.

4.2 Temporal RBAC extensions

Many extensions have been proposed to extend RBAC with temporal aspects. A review comparing several extensions can be found in [65].

The first RBAC extension which allowed for time-dependent accessibility is Temporal RBAC (TRBAC) [24]. TRBAC introduces the concepts of *role enabling* and *role disabling*. The permissions associated with a role can only be invoked if that role is enabled. Thus, a user can only use the permissions of an assigned role when it is enabled. Roles are enabled and disabled over time according to *temporal constraints*. TRBAC defines two types of temporal constraints: *periodic expressions* and *role triggers*. Periodic expressions state for a given set of time intervals which roles should be enabled during those intervals. When enabling or disabling certain roles, role triggers may define extra roles to enable or disable based on the current sets of enabled/disabled roles.

Two other notable extensions are generalized TRBAC (GTRBAC) [25] and GemRBAC+CTX [65]. GTRBAC is an extension of TRBAC. It adds duration/cardinality constraints on the role activation relation and constraints on the role-permission assignment relation. GemRBAC+CTX is an exceptionally expressive model which tries to cover most if not all the functionality offered by related models. The main difference between GTRBAC and GemRBAC+CTX is that the latter supports more complex temporal expressions representing one-off or repeating time intervals, permission enabling/disabling and constraints based upon spatial information. The authors of [65] themselves claim a difference is also that they support constraints on role-permission assignment while GTRBAC does not, however we believe this to be a small oversight as GTRBAC does support this type of constraint.

We believe the approach taken in TRBAC is a promising jumping-off point to extend RBÄC with temporal aspects. Therefore, we are going to try to rebase TRBAC onto

RBÄC in a way that is fitting for the domain of physical access control. We picked TRBAC over other more expressive models such as GTRBAC or GemRBAC+CTX because it offers enough functionality while keeping the complexity relatively low. Model expressiveness often comes at the cost of analyzability. In the case of GemRBAC+CTX for example, just making an access decision could take up to three seconds [65].

4.3 Combining RBÄC and TRBAC

Although TRBAC is a promising candidate, RBÄC and TRBAC do not combine well out of the box. Furthermore, not all ideas TRBAC introduces are appropriate for the domain of physical access control. We will therefore rebase the essential parts of TRBAC onto RBÄC.

4.3.1 Role Enabling / Disabling

The first roadblock when we try to rebase TRBAC onto RBÄC is the construct of role enabling/disabling. Intuitively, role enabling/disabling corresponds to stating that the role can only be used within a certain time frame. Although useful in some cases, we argue that this is often not what security officers want to express. Instead, we argue that instead they want to express that the groups of permissions associated with a role vary over time. In the case of physical access control, security officers rarely want to express that a role (e.g. employee) can only be used during working hours. Much more often, security officers want to express which areas this role should be able to access during working hours.

Furthermore, modelling an organizational role with varying (non-empty) permission sets over time through role-enabling/disabling requires multiple roles. For example, modelling that an employee has different permissions during and after working hours requires splitting the organization role of employee in a role which is enabled during working hours and a role which is enabled after working hours. This is against the spirit of RBÄC, as roles should represent abstractions over the set of users which are not tied to the implementation of the access control policy.

Thus, we propose to add time-dependent access control in RBÄC through granting and revoking a role's access to demarcations over time via temporal constraints. We will call this *demarcation granting* / *demarcation revoking*. In essence, this shifts the time restriction from the UR relation to the RD relation. Lastly, the concept of role enabling/disabling is fully disconnected from the concept of role hierarchies.

Roles can not become enabled/disabled through inheritance. However, in RBÄC, role hierarchies are defined such that demarcations granted to roles lower in the hierarchy are inherited by roles higher up in the hierarchy. We argue that since we are extending RBÄC, we should try to conserve the spirit behind the original extension as much as possible. Thus, we argue that demarcation granting/revoking should be connected to the concept of role hierarchies. Specifically, we argue that in GR-RBAC if a role lower in the hierarchy has access to a demarcation during a specific time period, this should still be inherited by roles higher up in the hierarchy during the same time period.

4.3.2 Temporal Constraints

Both types of temporal constraints introduced by TRBAC could be translated into temporal constraints on demarcation granting / revoking. However, although something akin to role triggers can be useful to improve policy conciseness and understandability, this type of constraint will not add any expressive power compared to something akin to periodic expressions while complicating the conflict detection and resolution process significantly. Although one can also specify inconsistent policies when just granting/revoking demarcations based on the time, inconsistencies are easily found since two statements will directly contradict each other and it can be resolved with a conflict resolution principle such as revocations taking precedence over grants.

To illustrate, consider the requirement that when role r_1 is granted demarcation d_1 it should also always be granted demarcation d_2. Instead of requiring possibly many rules which grant these demarcations to r_1 in the same temporal contexts, only a rule which states this dependency is required. However, this also allows to specify contradictions which are non-trivial to detect. Imagine that we also have another set of constraints which state that role r_1 should always be granted demarcation d_1, that when role r_1 is granted demarcation d_1 then role r_2 should also be granted d_1 and that when r_2 is granted d_1 then r_1 should not be granted d_1. This contradiction is non-trivial to detect nor to resolve. An algorithm to detect contradictions for role-enabling/disabling with periodic expressions/role triggers was presented in [24].

Another problem with this kind of temporal constraint is that it might be confusing for a security officer if something happens automatically when a demarcation is granted to or revoked from a role. Thus, it might be a better idea to force the security officer to specify when demarcations should be granted to roles solely based on the time and allow him/her to verify, in contrast to specify, that this requirement (i.e. an impli-

cation) holds.

To reduce the scope of our research, we have chosen to only support temporal constraints which grant or revoke demarcations based on the temporal context. We will call this type of constraint a *temporal grant rule*. We leave constraints which grant/revoke demarcations based upon which demarcations are being granted/revoked as possible future work.

4.3.3 Time Intervals

In TRBAC, time intervals are specified through periodic expressions. Periodic expressions are based on the concept of calendars [66] and define sets of time intervals based on a set of starting time instants and a duration in terms of a period (e.g. minutes, hours, days, etc.).

In most cases, security officers do not want to specify how long somebody should have access. Instead, they generally want to specify when somebody should have access. Although this might sound similar, expressing *when* somebody should have access while only being able to specify *how long* somebody should have access and vice versa is non-trivial for realistic policies as clock shifts have to be taken into account (e.g. daylight saving time, leap seconds, moving between time-zones, etc.). Additionally, finding all unique sets of intersecting periodic expressions also becomes non-trivial when taking clock shifts into account. Therefore, we argue that periodic expressions are not the right formalism to express time intervals in physical access control.

Therefore, we argue that a different formalism which allows security officers to specify repeating and one-off time intervals as a period between two points in time specified on a day of the week (e.g. Monday, Tuesday, etc.), a day of the month (the 1st of January, the 2nd of January, etc.), a day of the week/month (e.g. Monday 1st of January, Monday 2nd of January etc.) or a day of the year (Wednesday the 1st of January 2020, Thursday the 2nd of January 2020, etc.) would be more appropriate. Please note that by defining time intervals on the day of the week/month, it is possible to also express concepts such as "the 1st Monday of January" or "the 3rd Thursday of October" as these can be represented by a set of intervals on the day of the week/month (e.g. Monday on 1-7 January and Thursday on 15-21 October).

4.4 GR-RBAC Outline

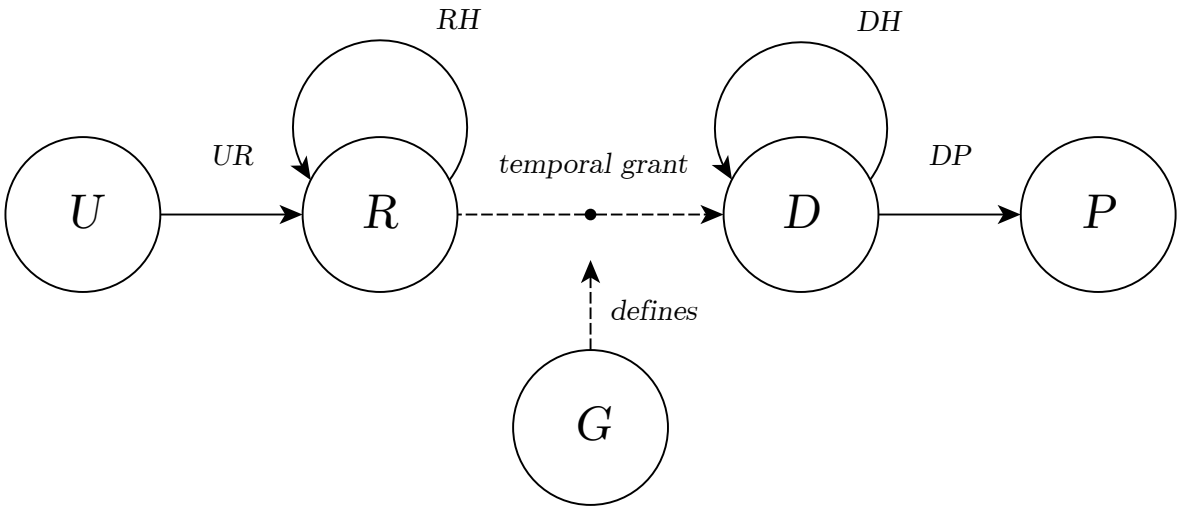


Figure 4.2: High-level overview of the GR-RBAC Authorization Model

Taking into account the presented roadblocks and how they might be overcome, we structured the new extension as follows. GR-RBAC spiritually extends RBÄC by adding a set of *temporal grant rules* G and a set of *temporal contexts* C. Together, they work as a kind of time table. We call the extension *Gregorian* RBAC as, when dealing with time, we make use of assumptions and simplifications which hold for the Gregorian calendar but which may not hold for other calendars.

Temporal grant rules grant/revoke roles access to demarcations based on the time. They may be given a priority to resolve conflicts. When they are conflicting, demarcation revoking takes precedence over demarcation granting [1]. To illustrate this and other new concepts, please consider the policy of the fictive ACME company. This policy is presented in Figure 4.3. For this policy, we could define the temporal grant rules shown in Figure 4.4.

Temporal contexts are named sets of *time ranges* which can represent concepts such as working hours, lunch breaks or holidays. We call the time ranges of a temporal context the instances of that context. They represent when the temporal context takes place. If any instance of a temporal context covers a moment in time, we also say that the temporal context as a whole covers that moment in time. Temporal grant rules grant/revoke roles access to demarcations during temporal contexts.

[1] This conflict-resolution approach is based on the denial-takes-precedence principle, which represents "the most conservative approach with respect to security" [67]. TRBAC also based their conflict-resolution strategy on this principle. More about conflict-resolution strategies for access control can be found in [68].

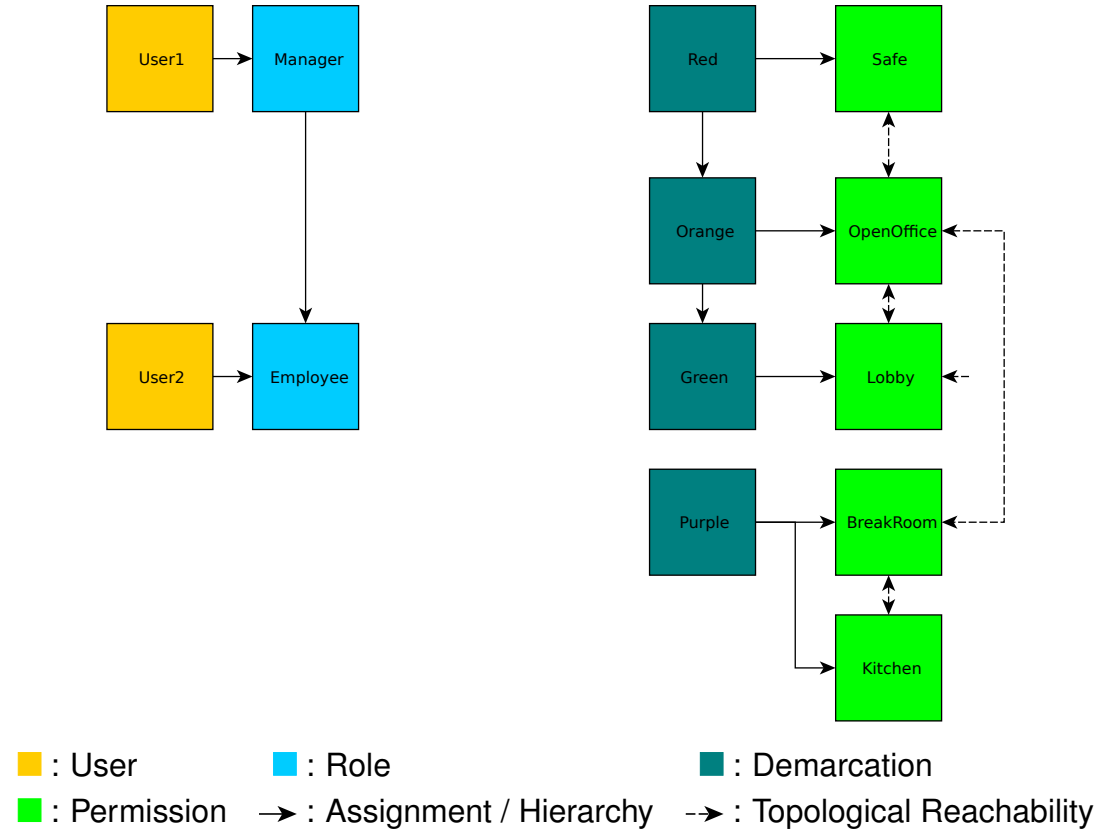


Figure 4.3: ACME GR-RBAC Authorization Model & Topology

Grant *Employee* **access to** *Green* **during** *Always* **with priority** *Low*
Grant *Employee* **access to** *Orange* **during** *Working Hours* **with priority** *Medium*
Grant *Employee* **access to** *Purple* **during** *Lunch Breaks* **with priority** *Medium*
Revoke *Employee* **access to** *Orange* **during** *Holidays* **with priority** *High*
Revoke *Employee* **access to** *Purple* **during** *Holidays* **with priority** *High*
Grant *Manager* **access to** *Red* **during** *Working Hours* **with priority** *Medium*

Figure 4.4: ACME Example Temporal Grant Rules

Temporal Context	Instances (Time Ranges)
Working Hours	{ Monday 8.00-17.00, [...], Friday 8.00 - 17.00 }
Lunch Breaks	{ Monday 12.00-13.00, [...], Friday 12.00 - 13.00 }
Holidays	{ 25 December 00.00 - 23.59 }
Always	{ Monday 0.00-23.59, [...], Sunday 0.00 - 23.59 }

Table 4.1: ACME Example Temporal Contexts

We assume the existence of a temporal context $Always$ covering all moments in time. This allows us to simplify various definitions regarding the authorization constraints and the computation of the set of scenarios later in this research. We believe this to be a reasonable assumption since we believe almost every realistic physical access control systems will have some behaviour which should always hold.

Time ranges are time intervals which can be specified on 4 levels: a day of the week (e.g Monday), day of the month (e.g. 25^{th} of December), day of the year (e.g. Monday 25^{th} of December 2023) or when a day of the week coincides with a day of the month (e.g. Monday 25^{th} of December). To illustrate, the temporal contexts used in the temporal grant rules shown in 4.4 could be specified as shown in Table 4.1. We assume $Always$ will always be defined as shown in this table.

From the set of temporal contexts, we can compute the set of *scenarios*. Scenarios are sets of temporal contexts which represent the occurring combinations of temporal contexts. In essence, the set of scenarios is the state-space of the authorization model. When we verify time-dependent authorization constraints, we check if it holds for all occuring scenarios.

From the set of scenarios and set the temporal grant rules, we can compute the temporal grant relation RSD. This relation captures which roles have access to which demarcations during which scenarios. From RSD we can in turn compute the access relation. This relation determines when users have access to which security zones. To also illustrate these concepts the scenarios, temporal grant relation and access relation of the ACME policy can be found in 4.5.

4.5 GR-RBAC Definitions

In this section, we will present the formal definitions capturing the new GR-RBAC model. The definitions will be illustrated using the ACME policy introduced in the

Scenario	Instances (Time Ranges)
{ Always, Working Hours, Lunch Breaks, Holidays }	{ Monday 25^{th} of December 12:00 - 13:00, [...], Friday 25^{th} of December 12:00 - 13:00 }
{ Always, Working Hours, Lunch Breaks }	{ Monday 1^{st} of January 12:00-13:00, Tuesday 1^{st} of January 12:00-13:00, [...] }
{ Always, Working Hours, Holidays }	{ Monday 25^{th} of December 08:00 - 11:59, Monday 25^{th} of December 13:01 - 17:00, [...], Friday 25^{th} of December 13:01 - 17:00 }
{ Always, Working Hours }	{ Monday 1^{st} of January 08:00-11:59, Monday 1^{st} of January 13:01-17:00, [...] }
{ Always, Holidays }	{ Monday 25^{th} of December 00:00 - 07:59, Monday 25^{th} of December 17:01 - 23:59, [...], Sunday 25^{th} of December 00:00 - 23:59 }
{ Always }	{ Monday 1^{st} of January 00:00-7:59, Monday 1^{st} of January 17:01-23:59, [...] }

Table 4.2: ACME Example Scenarios

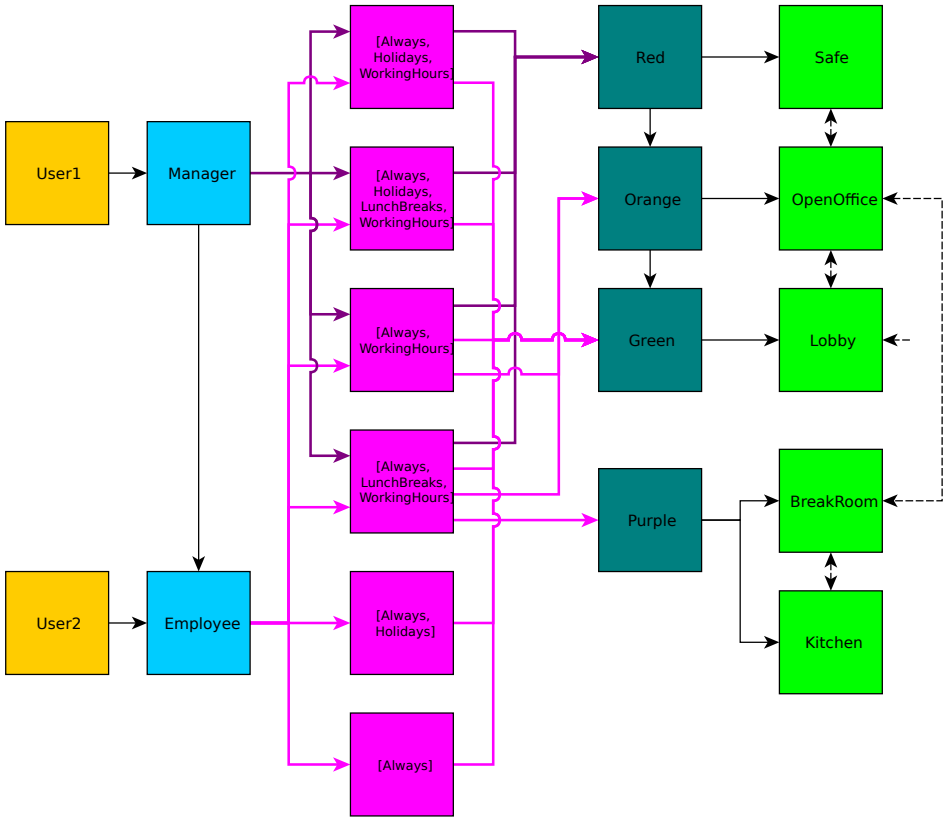


Figure 4.5: ACME GR-RBAC Auth. Model, Topology & Temporal Grant Relation

: User : Role : Demarcation

: Permission → : Assignment / Hierarchy --→ : Topological Reachability

: Scenarios →/→ : Temporal Assignments

previous section as a running example.

4.5.1 Temporal Grant Rules

Let us start with formalizing the definition of temporal grant rules. They are a central concept; temporal grant rules can be used to temporarily associate demarcations with roles. They grant/revoke a role's access to a demarcation during a temporal context with a certain priority.

Definition 4.5.1 *Temporal Grant Rules*
A temporal grant rule is a 5-tuple (a, r, d, c, p) *where:*

- $a \in \{grant, revoke\}$*: is an action,*
- $r \in R$*: is a role,*
- $d \in D$*: is a demarcation,*
- $c \in C$*: is a temporal context*
- $p \in \mathbb{N}^+$*: is a value representing a priority.*

Example 4.5.1 *ACME Temporal Grant Rules*
Consider the 6 temporal grant rules presented in Figure 4.4. They can be represented as:

$$\begin{aligned}
g_1 &= (grant, employee, green, always, 1) \\
g_2 &= (grant, employee, orange, working\ hours, 2) \\
g_3 &= (grant, employee, purple, lunch\ breaks, 2) \\
g_4 &= (revoke, employee, orange, holidays, 3) \\
g_5 &= (revoke, employee, purple, holidays, 3) \\
g_6 &= (grant, manager, red, working\ hours, 2)
\end{aligned}$$

4.5.2 Time Ranges

Next, let us formalize another new central concept: time ranges. They will be used to state when a temporal context takes place. Time ranges are time intervals specified on a day of the week/month/year or when a weekday coincides with a specific day of the month.

Please note that we use two assumptions regarding time in our definitions. We assume that every day of the week will eventually overlap with every day of the

month and that we can establish a set $Dates$ representing all valid dates in the Gregorian calendar. Furthermore, please note that although here we have chosen to represent time up to the granularity of minutes, this could be replaced by any other granularity without issue as computations are only performed with the lower and upper bounds of the intervals (and no computations iterates over all the moments captured by interval).

Definition 4.5.2 *Time Range*
A time range is a 5-tuple (I, e, d, m, y) *where:*

- I*: is a integer interval representing a time-of-day interval in minutes,*
- $e \in \{Mon, Tue, Wed, Thu, Fri, Sat, Sun\} \cup \{\_\}$*: represent a weekday or a wildcard,*
- $d \in \mathbb{N}^+ \cup \{\_\}$*: represents the* n^{th} *day of a month or a wildcard,*
- $m \in \{Jan, Feb, Mar, Apr, [...], Dec\} \cup \{\_\}$*: represents a month or a wildcard,*
- $y \in \mathbb{N} \cup \{\_\}$*: represents a year or a wildcard.*

A time range is considered valid if $I \subseteq [0, 1439]$, $|I| \geq 1$ *and it belongs to one of the following sets:*

- $\mathcal{T}_w = \{(I, e, \_, \_, \_) \mid e \in \{Mon, ..., Sun\}\}$*, representing all valid time intervals on a day of the week (e.g. Monday),*
- $\mathcal{T}_m = \{(I, \_, d, m, \_) \mid 1 \leq d \leq days(m)\}$*, representing all valid time intervals on a day of month (e.g. 25*th *of December),*
- $\mathcal{T}_{wm} = \{(I, e, d, m, \_) \mid e \in \{Mon, ..., Sun\}, 1 \leq d \leq days(m)\}$*, representing all valid time intervals when a weekday coincides with a specific day of the month (e.g. Monday 25*th *of December),*
- $\mathcal{T}_y = \{(I, e, d, m, y) \mid (e, d, m, y) \in Dates\}$*, representing all valid time intervals on a day of the year (e.g. Monday 25*th *of December 2023).*

with $days = \{(Jan, 31), (Feb, 29), (Mar, 31), ..., (Dec, 31))$ *a function giving the maximum days of a month and* $Dates = \{..., (Thu, 1, Jan, 1970), (Fri, 2, Jan, 1970), ...\}$ *a set representing all valid dates in the Gregorian calendar.*

The set of all valid time ranges is given by $\mathcal{T} = \mathcal{T}_w \cup \mathcal{T}_m \cup \mathcal{T}_{wm} \cup \mathcal{T}_y$

Definition 4.5.3 *Time Range Intersection*

Let t_1 and t_2 be two valid time ranges. We define the intersection $t_1 \cap t_2$ as:

$$t_1 \cap t_2 = \begin{cases} (I_1 \cap I_2, e_1, d_2, m_2, y_2) & \textit{if } t_1 = (I_1, e_1, \_, \_, \_), t_2 = (I_2, e_2, d_2, m_2, y_2) \\ & \wedge\ I_1 \cap I_2 \neq \{\}\ \wedge\ (e_1 = e_2 \vee e_2 = \_) \\ (I_1 \cap I_2, e_2, d_1, m_1, y_2) & \textit{if } t_1 = (I_1, \_, d_1, m_1, \_), t_2 = (I_2, e_2, d_2, m_2, y_2) \\ & \wedge\ I_1 \cap I_2 \neq \{\}\ \wedge\ (d_1 = d_2 \vee d_2 = \_) \\ & \wedge\ (m_1 = m_2 \vee m_2 = \_) \\ (I_1 \cap I_2, e_1, d_1, m_1, y_2) & \textit{if } t_1 = (I_1, e_1, d_1, m_1, \_), t_2 = (I_2, e_2, d_2, m_2, y_2) \\ & \wedge\ I_1 \cap I_2 \neq \{\}\ \wedge\ (e_1 = e_2 \vee e_2 = \_) \\ & \wedge\ (d_1 = d_2 \vee d_2 = \_)\ \wedge\ (m_1 = m_2 \vee m_2 = \_) \\ (I_1 \cap I_2, e_1, d_1, m_1, y_1) & \textit{if } t_1 = (I_1, e_1, d_1, m_1, y_1), t_2 = (I_2, e_2, d_2, m_2, y_2) \\ & \wedge\ I_1 \cap I_2 \neq \{\}\ \wedge\ (e_1 = e_2 \vee e_2 = \_) \\ & \wedge\ (d_1 = d_2 \vee d_2 = \_)\ \wedge\ (m_1 = m_2 \vee m_2 = \_) \\ & \wedge\ (y_1 = y_2 \vee y_2 = \_) \\ ([0, 0), \_, \_, \_, \_) & \textit{otherwise} \end{cases}$$

Definition 4.5.4 *Time Range Subrange Relation*
Let t_1 and t_2 be two valid time ranges. We define the subrange relation $t_1 \subseteq t_2$ as:

$$t_1 \subseteq t_2 \Leftrightarrow t_1 \cap t_2 = t_1$$

Definition 4.5.5 *Time Instant*
A time instant is a time range $(I, e, d, m, y) \in \mathcal{T}_y$ where $|I| = 1$. $\mathcal{T}_\tau$ is the set of all time instants.

Example 4.5.2 *Time Range Examples*
To illustrate the previously introduced definitions, let us define 4 valid time ranges t_1, t_2, t_3, t_4:

$$\begin{aligned} t_1 &= ([480, 1020], Mon, \_, \_, \_) \\ t_2 &= ([720, 780], Mon, \_, \_, \_) \\ t_3 &= ([0, 1439], \_, 25, Dec, \_) \\ t_4 &= ([720, 720], Mon, 25, Dec, 2023) \end{aligned}$$

Here, t_1 represent the time range from 8am to 5pm on Mondays, t_2 represent the time range from 12am to 1pm on Mondays, t_3 represent the 25th of December and t_4 is a time instant which represents 12:00am on Monday the 25th of December 2023.

For t_1, t_2, t_3 and t_4, it holds that: $t_2 \subseteq t_1$, $t_4 \subseteq t_1$, $t_4 \subseteq t_2$, $t_4 \subseteq t_3$

$t_1 \cap t_2 = ([720, 780], Mon, \_, \_, \_) = t_2$
$t_1 \cap t_3 = ([480, 1020], Mon, 25, Dec, \_)$
$t_1 \cap t_4 = ([720, 720], Mon, 25, Dec, 2023) = t_4$
$t_1 \cap t_2 \cap t_3 = ([720, 780], Mon, 25, Dec, \_)$
$t_1 \cap ([480, 1020], Tue, \_, \_, \_) = ([0, 0), \_, \_, \_, \_)$

Example 4.5.3 *ACME Time Range Sets*
Consider the 4 temporal contexts $working\ hours$, $lunch\ break$, $holidays$ and $always$ presented in Table 4.1. The sets of time ranges which represent when these temporal contexts take place can be represented as:

$$
\begin{aligned}
T_{WH} &= \{([480, 1020], Mon, \_, \_, \_), [...], ([480, 1020], Fri, \_, \_, \_)\} \\
T_{LB} &= \{([720, 780], Mon, \_, \_, \_), [...], ([720, 780], Fri, \_, \_, \_)\} \\
T_{H} &= \{([0, 1439], \_, 25, Dec, \_)\} \\
T_{A} &= \{([0, 1439], Mon, \_, \_, \_), [...], ([0, 1439], Sun, \_, \_, \_)\}
\end{aligned}
$$

4.5.3 Authorization Model

With time ranges and temporal grant rules formally defined, they can now be put together in an authorization model. We extend RBÄC and add:

- a set of temporal contexts C,
- a function γ which maps temporal contexts to a set of time ranges which represent when the temporal context takes place,
- a set of temporal grant rules G.

We will call this extension Gregorian RBAC (GR-RBAC).

Definition 4.5.6 *GR-RBAC Policy Authorization Model*
An authorization model $\mathcal{M}$ is a 11-tuple $(U, R, D, P, O, UR, DP, PO, C, \gamma, G)$ where

- *U, R, D, P,O: are sets of users, roles, demarcations, permissions, objects*
- *$UR \subseteq U \times R$: is a user-role assignment relation,*
- *$DP \subseteq D \times P$: is a demarcation-permission assignment relation,*
- *$PO \subseteq P \times O$: is a permission-object assignment relation,*
- *C: is a set of temporal contexts,*

- $\gamma : C \to \mathcal{P}(\mathcal{T})$*: is a function mapping each temporal context to its instances,*
- G*: is a set of temporal grant rules.*

For an authorization model $\mathcal{M}$ we write $U_{\mathcal{M}}$ to indicate $\mathcal{M}$'s set of users, $R_{\mathcal{M}}$ to indicate $\mathcal{M}$'s set of roles and so on, unless it is clear from its context.

Definition 4.5.7 *Role-Hierarchy*
A role-hierarchy is a partial order $RH \subseteq R \times R$. We write $r_i > r_j$ for $(r_i, r_j) \in RH$ meaning that r_i inherits the demarcations granted to r_j. In this case, we say that r_i is a senior role of r_j and r_j is a junior role of r_i.

Definition 4.5.8 *Demarcation-Hierarchy*
A demarcation-hierarchy is a partial order $DH \subseteq D \times D$. We write $d_i > d_j$ for $(d_i, d_j) \in DH$ meaning that d_i inherits the permissions assigned to d_j. In this case, we say that d_i is a superdemarcation of d_j and d_j is a subdemarcation of d_i.

Please note that since the hierarchies are defined as partial orders, cyclic inheritance is not possible.

Example 4.5.4 *ACME GR-RBAC Policy Authorization Model*
Consider the policy presented in Figure 4.5, it can be represented as a GR-RBAC policy $\mathcal{M}_{ACME} = (U_{ACME}, R_{ACME}, D_{ACME}, P_{ACME}, O_{ACME}, UR_{ACME}, DP_{ACME}, PO_{ACME}, C_{ACME}, \gamma_{ACME}, G_{ACME})$ with:

$$
\begin{aligned}
U_{ACME} &= \{user1, user2\} \\
R_{ACME} &= \{employee, manager\} \\
RH_{ACME} &= \{(manager, employee)\} \\
D_{ACME} &= \{red, orange, green, purple\} \\
DH_{ACME} &= \{(red, orange), (orange, green)\} \\
P_{ACME} &= \{p1, p2, p3, p4, p5\} \\
O_{ACME} &= \{safe, open\ office, lobby, breakroom, kitchen\} \\
UR_{ACME} &= \{(user1, manager), (user2, employee)\} \\
DP_{ACME} &= \{(red, safe), (orange, open\ office), (green, lobby)(purple, breakroom) \\
&\quad (purple, kitchen)\} \\
PO_{ACME} &= \{(p1, safe), (p2, open\ office), (p3, lobby), (p4, breakroom), (p5, kitchen)\} \\
C_{ACME} &= \{working\ hours, lunch\ break, holidays, always\} \\
\gamma_{ACME} &= \{(working\ hours, T_{WH}), (lunch\ break, T_{LB}), (holidays, T_H), (always, T_A)\} \\
G_{ACME} &= \{g_1, g_2, g_3, g_4, g_5, g_6\}
\end{aligned}
$$

To improve the readability, we will abbreviate the temporal contexts $working\ hours$, $lunch\ break$, $holidays$ and $always$ as c_{WH}, c_{LB}, c_H and c_A in most future examples.

4.5.4 Scenarios

To allow for time-dependent accessibility, the access relation is redefined such that the newly defined temporal contexts and the temporal grant rules are taken into account. This is done on the basis of scenarios. Scenarios are sets of temporal contexts which represent the occurring combinations of temporal contexts. Similar to temporal contexts, the time ranges which represent when the scenario takes place are called the instances of that scenario.

Definition 4.5.9 *Covers*
Given a time range t, a set of temporal contexts Q and a temporal context instance function f, the function $covers : \mathcal{T} \times \mathcal{P}(C) \times (C \to \mathcal{P}(\mathcal{T})) \to \mathcal{P}(C)$ returns all temporal contexts in Q which cover t.

$$covers(t, Q, f) \quad = \quad \{\, c \in Q \mid \exists t' \in f(c), t \subseteq t' \}$$

Definition 4.5.10 *Scenarios*
Given the set of temporal contexts Q and the temporal context instance function f of an authorization model, the function $scenarios : \mathcal{P}(C) \times (C \to \mathcal{P}(\mathcal{T})) \to \mathcal{P}(C)$ returns the set of scenarios of the authorization model.

$$scenarios(Q, f) \quad = \quad \{\, S \in \mathcal{P}(Q) \mid \exists t \in \mathcal{T}_\tau, covers(t, Q, f) = S \}$$

Definition 4.5.11 *Scenario Instances*
Given a scenario S, a set of temporal contexts Q and a temporal context instance function f of an authorization model, the function $instances : \mathcal{P}(C) \times (C \to \mathcal{P}(\mathcal{T})) \to \mathcal{P}(\mathcal{T}_\tau)$ returns the instances of S.

$$instances(S, Q, f) \quad = \quad \{\, t \in \mathcal{T}_\tau \mid covers(t, Q, f) = S \}$$

Example 4.5.5 *ACME Scenarios*
$scenarios(C_{ACME}, \gamma_{ACME}) = \{\{c_{WH}, c_{LB}, c_H, c_A\}, \{c_{WH}, c_{LB}, c_A\}, \{c_{WH}, c_H, c_A\}, \{c_{WH}, c_A\}, \{c_H, c_A\}, \{c_A\}\}$

4.5.5 Access Relation

We can now define the *access relation*. This relations states who (user) has access to what (security zone) during which moments in time (scenarios). The difference between RBÄC's and GR-RBAC's access relation is that roles are now granted demarcations depending on the temporal contexts. This is captured in the *temporal grant relation*.

Definition 4.5.12 *Temporal Grant Relation*
For a authorization model $\mathcal{M}$, we define its temporal grant relation as $RSD \subseteq R \times \mathcal{S} \times D$ as:

$$RSD = \{(r, S, d) \in R \times \mathcal{S} \times D \mid grant_{max} > revoke_{max}\}$$

with $\mathcal{S} = scenarios(C, \gamma)$
$grant_{max} = max(0, max(\{\ p \mid (grant, r, d, c, p) \in G, c \in S\ \}))$
$revoke_{max} = max(0, max(\{\ p \mid (revoke, r, d, c, p) \in G, c \in S\ \}))$

Example 4.5.6 *ACME Temporal Grant Relation*

$$\begin{aligned} RSD_{ACME} &= \{employee\} \times \mathcal{S} \times \{green\} \\ &\cup \{employee\} \times \{\{c_{WH}, c_{LB}, c_A\}, \{c_{WH}, c_A\}\} \times \{orange\} \\ &\cup \{employee, \{c_{WH}, c_{LB}, c_A\}, purple\} \\ &\cup \{manager\} \times \{\{c_{WH}, c_{LB}, c_H, c_A\}, \{c_{WH}, c_H, c_A\}, \\ &\quad \{c_{WH}, c_{LB}, c_A\}, \{c_{WH}, c_A\}\} \times \{red\} \end{aligned}$$

with $\mathcal{S} = scenarios(C_{ACME}, \gamma_{ACME})$

Definition 4.5.13 *Access Relation*
For a authorization model $\mathcal{M}$, we define its access relation $USO \subseteq U \times \mathcal{S} \times O$ such that $(u, S, o) \in USO$ if there exists two roles $r, r' \in R$, two demarcations $d, d' \in D$ and a permission $p \in P$ such that the following conditions are fulfilled:

- $(u, r) \in UR$, *i.e.: user u is assigned role r,*
- $r \geq r'$, *i.e.:* $r = r'$ *or* r *is senior role of* r',
- $(r', S, d) \in RSD$, *i.e.: role* r *is granted demarcation* d *during scenario* S,
- $d \geq d'$, *i.e.:* $d = d'$ *or* d *is a superdemarcation of* d',
- $(d', p) \in DP$, *i.e.: permission* p *is part of demarcation* d',
- $(p, o) \in PO$, *i.e.: permission* p *gives access to object* o.

where $\mathcal{S} = scenarios(C, \gamma)$

Example 4.5.7 *ACME Access Relation*

$$\begin{aligned} USO_{ACME} &= \{user1, user2\} \times \mathcal{S} \times \{lobby\} \\ &\cup \{user1, user2\} \times \{\{c_{WH}, c_{LB}, c_A\}, \{c_{WH}, c_A\}\} \times \{lobby, open\ office\} \\ &\cup \{user1, user2\} \times \{\{c_{WH}, c_{LB}, c_A\}\} \times \{breakroom, kitchen)\} \end{aligned}$$

$$\cup \quad \{user1\} \times \{\{c_{WH}, c_{LB}, c_H, c_A\}, \{c_{WH}, c_H, c_A\}, \{c_{WH}, c_{LB}, c_A\}, \{c_{WH}, c_A\}\} \times \{lobby, open\ office, safe\}$$

with $\mathcal{S} = scenarios(C_{ACME}, \gamma_{ACME})$

Chapter 5

Site Access Control System Model

To account for authorization constraints dependent on the access control system topology and the authentication model in a way that is suitable for incremental verification, we propose to link them with the authorization model by looking for a shared concept which exists or can be derived from each.

In this chapter, we will define this new formalism, which we will call the *site access control system* (SACS) model. It will consists of a GR-RBAC based authorization model, an access control system topology model and a simplified authentication model. We propose to link them in the SACS model through the shared concept of security zones as this is a central concept:

- authorization models describe who should have access to which security zones depending on the time,
- the access control system topology model describes how security zones are interconnected and connected with the outside,
- authentication models describe the obligations to access a security zone depending on the time.

5.1 Context Models

We only capture the information in the access control topology model and the authentication model which we deem essential to verify the key context-dependent invariants presented in the introduction. These invariants were:

- users can never get trapped, i.e. users should be able to leave the collection of areas under access control.

- users can always enter the security zones he/she has the permission to access, i.e. users should always be able invoke all granted permissions.

Specifically, we will capture the how security zones are interconnected and which security zones allow users to enter/leave the areas under access control in the access control system topology model and the effects of the authentication model on the accessibility of the security zones.

5.1.1 Acces Control System Topology Model

From the access control system topology, we capture the how security zones are interconnected and which security zones allow users to enter/leave the areas under access control. Although other topological properties can play a part in the invariants being violated, we deem these properties to play the largest role.

Definition 5.1.1 *Access Control System Topology Model*
An access control system topology model $\mathcal{V}$ is a 3-tuple (O, I, δ) where

- *O: is a set of security zones,*
- *$I \subseteq O$: is the subset of the security zones which can be accessed from outside the security system and allows users to leave the security system,*
- *$\delta \subseteq O \times O$: is a relation representing (direct) reachability between security zones.*

For an access control system topology model $\mathcal{V}$ we write $O_{\mathcal{V}}$ to indicate $\mathcal{A}$'s set of security zones, $I_{\mathcal{V}}$ to indicate $\mathcal{V}$'s set of security zones which can be accessed from outside the security system contexts and $\delta_{\mathcal{V}}$ to indicate the reachability between $\mathcal{V}$'s security zones, unless clear from its context.

Example 5.1.1 *ACME Access Control System Toplogy*
Consider the policy presented in Figure 4.5, the access control system topology can be represented as a $\mathcal{V}_{ACME} = (O_{ACME}, I_{ACME}, \delta_{ACME})$ with:

$$
\begin{aligned}
O_{ACME} &= \{safe, open\ office, lobby, breakroom, kitchen\} \\
I_{ACME} &= \{lobby\} \\
\delta_{ACME} &= \{(safe, open\ office), (open\ office, safe), (open\ office, lobby), \\
&\quad (lobby, open\ office), (open\ office, breakroom), (breakroom, open\ office), \\
&\quad (breakroom, kitchen), (kitchen, breakroom)\}
\end{aligned}
$$

5.1.2 Authentication Model

From the authentication model, we capture the effects of the authentication model on the accessibility of the security zones. This means that we abstract away from specific obligations such as the requirements (e.g. that users should enter a password or show a badge). Instead, we capture if for any given moment a security zone is:

- unlocked, meaning it is publicly accessible without authentication.
- locked, meaning it is inaccessible regardless of authentication.
- protected, meaning it is protected with authentication measures and thus can only be entered with the appropriate permissions.

We propose to define these statuses similarly to how we defined the access relation in the authorization model: through rules which state if a zone should be unlocked/locked/protected during a certain temporal context with a given priority. We chose to define authentication on the level of security zones instead of their connects between them as we also defined authorization on the level of security zones (instead of on the connections between them). As the most common scenario is that areas are protected with authentication measures, we say that the default status is that a security zone is protected. As a conflict resolution policy, we take that the assignment of the locked status takes precedence over that of the protected status, which in turn takes precedence over that of the unlocked status.

Definition 5.1.2 *Temporal Authentication Rules*
A temporal authentication rule is a 4-tuple (a, o, t, p) *where:*

- $a \in \{locked, unlocked, protected\}$*: is a status,*
- $o \in O$*: is a security zone,*
- $c \in C$*: is a temporal context,*
- $p \in \mathbb{N}$*: is a value representing a priority.*

Definition 5.1.3 *Authentication Model*
An authentication model $\mathcal{A}$ *is a 4-tuple* (O, C, γ, A) *where*

- O*: is a set of security zones,*
- C*: is the set of temporal contexts,*
- $\gamma : C \rightarrow \mathcal{P}(\mathcal{T})$*: is the temporal context instance function,*
- A*: is a set of temporal authentication rules.*

For an authentication model $\mathcal{A}$ we write $O_{\mathcal{A}}$ to indicate $\mathcal{A}$'s set of security zones, $C_{\mathcal{A}}$ to indicate $\mathcal{A}$'s set of temporal contexts and so on, unless clear from its context.

Definition 5.1.4 *Temporal Authentication Status*
Given a scenario S, a security zone o and a set of authentication rules A, the function $status : \mathcal{P}(\mathcal{S}) \times O \times \mathcal{P}(A) \rightarrow \{locked, unlocked, protected\}$ computes the authentication status of o during S given A.

$$status(S, o, A) = \begin{cases} locked & \text{if } locked_{max} \geq protected_{max} \\ & \wedge\ locked_{max} \geq unlocked_{max} \\ & \wedge\ \exists (a, o, c, p) \in A, c \in S \\ unlocked & \text{if } unlocked_{max} > locked_{max} \\ & \wedge\ unlocked_{max} > protected_{max} \\ & \wedge\ \exists (a, o, c, p) \in A, c \in S \\ protected & \text{otherwise} \end{cases}$$

with $locked_{max} = max(0, max(\{\ p \mid (locked, o, c, p) \in A, c \in S\}))$
$protected_{max} = max(0, max(\{\ p \mid (protected, o, c, p) \in A, c \in S\}))$
$unlocked_{max} = max(0, max(\{\ p \mid (unlocked, o, c, p) \in A, c \in S\}))$

Example 5.1.2 *ACME Authentication Model*
Consider the policy presented in Figure 4.5, a possible authentication model could be $\mathcal{A}_{ACME} = (O_{ACME}, C_{ACME}, \gamma_{ACME}, A_{ACME})$ with O_{ACME}, C_{ACME}, γ_{ACME} as defined in 4.5.6 and

$$A_{ACME} \quad = \quad \{(unlocked, lobby, c_{WH}, 3), (locked, safe, c_H, 3)\}$$

Please note that these example temporal authentication rules correspond to something which, according to Nedap's experience, is not uncommon in access control systems. Quite often, the lobby of a company is publicly accessible during working hours. Furthermore, there are systems in which the authentication devices of a security zone are sometimes turned off (e.g. during weekends or holidays). This implies that somebody can only enter the security zone by overriding the system (e.g via a key).

Example 5.1.3 *ACME Temporal Authentication Statuses*
To illustrate Definition 5.1.4, consider the authentication model presented in Figure 5.1.2. Then, the following holds:

$$\forall o \in O, status(o, \{c_A\}, A_{ACME}) = protected$$

$$\begin{aligned} status(lobby, \{c_{WH}, c_{LB}, c_H, c_A\}, A_{ACME}) &= unlocked \\ status(open\ office, \{c_{WH}, c_{LB}, c_H, c_A\}, A_{ACME}) &= protected \\ status(safe, \{c_{WH}, c_{LB}, c_H, c_A\}, A_{ACME}) &= locked \end{aligned}$$

5.2 SACS Model

Putting everything together, we define the new SACS model as follows:

Definition 5.2.1 *SACS Model*
A site access control system model $\mathcal{P}$ is a 3-tuple $(\mathcal{M}, \mathcal{A}, \mathcal{V})$ with:

- *$\mathcal{M}$: an authorization model,*
- *$\mathcal{A}$: an authentication model,*
- *$\mathcal{V}$: an access control system topology model.*

The models should share the same set of security zones, temporal contexts and the temporal context instance function. Otherwise said, it should hold that $O_{\mathcal{M}} = O_{\mathcal{A}} = O_{\mathcal{V}}$, $C_{\mathcal{M}} = C_{\mathcal{A}}$ and $\gamma_{\mathcal{M}} = \gamma_{\mathcal{A}}$.

For a site access control system model $\mathcal{P}$, we write $U_{\mathcal{P}}$ to indicate its authorization model $\mathcal{M}$'s set of users $U_{\mathcal{M}}$, $RSD_{\mathcal{P}}$ to indicate its authorization model $\mathcal{M}$'s temporal grant relation RSD and so on for all entities and relations of $\mathcal{M}$, $\mathcal{A}$ and $\mathcal{V}$ unless it is clear from its context.

Example 5.2.1 *ACME SACS Model*
Putting everything together, ACME's access control system can be modeled as $\mathcal{P}_{ACME} = (\mathcal{M}_{ACME}, \mathcal{A}_{ACME}, \mathcal{V}_{ACME})$ with:

$\mathcal{M}_{ACME}$ as defined in Example 4.5.6, $\mathcal{A}_{ACME}$ as defined in Example 5.1.2 and $\mathcal{V}_{ACME}$ as defined in Example 5.1.1.

Chapter 6

Authorization Constraints

The number of authorization constraint types that can be specified and verified is only limited by the imagination of the security officer. In our research, we will focus on incrementally verifying a predefined set of authorization constraint types. This set will consist of common authorization constraints from the literature, the new authorization constraints defined together with security experts from Nedap and the context-dependent authorization constraints needed to verify the key invariants presented in the introduction.

In this section, we will propose some definitions regarding authorization constraints and the authorization constraint types to enable and facilitate their verification.

6.1 Groups

We propose to characterize authorization constraints as being policy-dependent or policy-independent. Policy-dependent authorization constraints refer to authorization constraints on specific elements of the authorization model. Policy independent authorization constraints refer to constraints on the structure of the authorization model, independent of specific elements.

6.2 Policy-dependent authorization constraints

As we presented in the background, the four common types of authorization constraints are: prerequisite, separation of duty (SoD), binding of duty (BoD) and cardinality constraints. These are all policy-dependent authorization constraints as they are specified on the basis of specific elements from the model.

Formally, we specify these constraints as shown below. These definitions are loosely inspired by the work presented in [69].

Definition 6.2.1 *Prerequisite Constraint*
A prerequisite constraint for an SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$ *is represented as* $PRE((e_p, e_d), E, c)$*, where:*

- $(e_p, e_d) \in (U^2 \cup R^2 \cup D^2 \cup P^2)$*: the prerequisite condition,*
- $E \in \{U, R, D, P\}$*: the set of entities for which the constraint should hold,*
- $c \in C$*: the temporal context during which the constraint should hold.*

with $e_p, e_d \notin E$

A prerequisite constraint $PRE((e_p, e_d), E, c)$ *is satisfied if* $\forall S \in \{Q \in \mathcal{S} \mid c \in Q\}$ *it holds that:*

$$\forall e \in E, (e, e_d) \in REL_S^* \rightarrow (e, e_p) \in REL_S^*$$

with $RD_Q = \{(r, d) \mid \exists (r, Q, d) \in RSD\}$
$REL_Q = UR \cup RD_Q \cup DP \cup RH \cup DH$
$REL_Q^* = symmetric\ closure\ of\ the\ transitive\ closure\ of\ REL_Q$
$\mathcal{S} = scenarios(C, \gamma)$

Definition 6.2.2 *Separation of Duty (SoD) Constraint*
A separation of duty constraint for an SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$ *is represented as* $SoD((e_1, e_2), E, t)$*, where:*

- $(e_1, e_2) \in (U^2 \cup R^2 \cup D^2 \cup P^2)$*: are the conflicting entities,*
- $E \in \{U, R, D, P\}$*: the set of entities for which the constraint should hold,*
- $c \in C$*: the temporal context during which the constraint should hold.*

with $e_1, e_2 \notin E$

A SoD constraint $SoD((e_1, e_2), E, t)$ *is satisfied if* $\forall S \in \{Q \in \mathcal{S} \mid c \in Q\}$ *it holds that:*

$$\forall e \in E, \neg((e, e_1) \in REL_S^* \wedge (e, e_2) \in REL_S^*)$$

with REL_S^* *as defined in 6.2.1*
and $\mathcal{S} = scenarios(C, \gamma)$*.*

Definition 6.2.3 *Binding of Duty (BoD) Constraint*
A binding of duty constraint for an SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$ *is represented as* $BoD((e_1, e_2), E, t)$*, where:*

- $(e_1, e_2) \in (U^2 \cup R^2 \cup D^2 \cup P^2)$*: are two model entities called the bound entities,*

- $E \in \{U, R, D, P\}$*: the set of entities for which the constraint should hold,*
- $c \in C$*: represent the temporal context during which the constraint should hold.*

with $e_1, e_2 \notin E$

A BoD constraint $BoD((e_1, e_2), E, t)$ *is satisfied if* $\forall S \in \{Q \in \mathcal{S} \mid c \in Q\}$ *it holds that:*

$$\forall e \in E, (e, e_1) \in REL_S^* \iff (e, e_2) \in REL_S^*$$

with REL_S^* *as defined in 6.2.1*
and $\mathcal{S} = scenarios(C, \gamma)$*.*

Definition 6.2.4 *Cardinality Constraint*
A cardinality constraint for an SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$ *is represented as* $CRD(e_b, u, E, t)$*, where:*

- $e_b \in (U \cup R \cup D \cup P)$*: represents the bound entity,*
- $u \in \mathbb{N}$ *represents the upper bound,*
- $E \in \{U, R, D, P\}$*: represents the set of entities for which the constraint should hold,*
- $c \in C$*: represent the temporal context during which the constraint should hold.*

with $e_b \notin E$

A cardinality constraint $CRD(e_b, u, B, t)$ *is satisfied if* $\forall S \in \{Q \in \mathcal{S} \mid c \in Q\}$ *it holds that:*

$$|\{(e, e_b) \in REL_S^* \mid e \in E\}| \leq u$$

with REL_S^* *as defined in 6.2.1*
and $\mathcal{S} = scenarios(C, \gamma)$*.*

Example 6.2.1 *ACME Constraints*

$$\begin{aligned}
prereq_1 &= PRE((manager, employee), U_{ACME}, always) \\
prereq_2 &= PRE((employee, manager), U_{ACME}, always) \\
prereq_3 &= PRE((kitchen, lobby), R_{ACME}, always) \\
sod_1 &= SoD((manager, employee), U_{ACME}, always) \\
bod_1 &= BoD((manager, employee), U_{ACME}, always) \\
bod_2 &= BoD((kitchen, lobby), R_{ACME}, c_{WH}) \\
card_1 &= CARD(safe, 1, U_{ACME}, always)
\end{aligned}$$

$prereq_2$ is not satisfied since not every employee is also a manager. sod_1 is not satisfied since $user_1$ is both a manager and a employee through inheritance. bod_1 is not satisfied since $user_2$ is only a employee. The rest of the authorization constraints are satisfied

Definition 6.2.5 *Static / Dynamic Constraints*
We say that an authorization constraint is static if either its satisfaction is independent of the associated temporal context or if the associated temporal context equals $always$. Otherwise, we say it is dynamic.

We employ the naming convention `(static/dynamic)` ␣ `level` ␣ `constraint name` to distinguish between subtypes of the four previously defined authorization constraint types. Here, level denotes the combination of the groups of elements of which the assignment or (transitive) grant relation is constrained. For example, $prereq_1$ would be a static user-role prerequisite constraint and bod_2 would be a a dynamic role-permission SoD constraint. We will use this naming convention when evaluating the performance of our incremental approach.

A limit of how we have defined the authorization constraints is that it is no distinction is made if the relationships occur because of direct assignments/grants or because of role-/demarcation-inheritance. Take the case of sod_1; it would be interesting to make a distinction between if a role has been granted directly (which might be a mistake) or indirectly (which happens because of inheritance). Another limit is that it is not possible to define exceptions. Sometimes constraints might be broken for valid reasons. Both would be interesting additions. However, we have decided to keep the authorization constraints relatively simple for this initial exploration into incremental verification.

6.3 Policy-independent authorization constraints

Together with security experts from Nedap, we established a list of policy-independent authorization constraints. These authorization constraints serve as *policy smell indicators*. They check for, what we will call, *policy smells*. The idea behind policy smells is similar to that of code smells. Policy smells are not direct flaws but could indicate a possible deeper structural problem. To illustrate, consider the "unused role" smell: a role which is not assigned or inherited. The presence of this smell could be an indicator that this role should not exist. However, the presence of this smell might be justified because of administrative reasons, e.g. there currently are no people occupying certain roles within the organization, or it makes the policy specification more readable. An overview of all policy smell indicators we will check for together

with an informal description of what they detect can be found in Table 6.1.

Policy Smell	**Description**
Unused Role Smell	a role that is never assigned to a user or inherited by another role
Unused Demarcation Smell	a demarcation that is never granted/revoked with a temporal grant rule or inherited by another demarcation
Unused Permission Smell	a permission that is never assigned to a demarcation
Zombie Demarcation Smell	a permission that is never granted to a role or inherited by another demarcation
Zombie Permission Smell	a permission that is never (transitively) granted to a role
God User Smell	a user that is always granted all permissions
God Role Smell	a role that is always granted all permissions
Ignored Role Inheritance	a role that is assigned to a user and of which a senior role is also granted to a user
Ignored Demarcation Inheritance	a demarcation that is granted to a role and of which a superdemarcation is also granted to a role

Table 6.1: Policy Smells

The context-dependent invariants which state that users should never be trapped and granted permissions should always be invocable can also be enforced with policy-independent authorization constraints. They do however depend on the context, namely the access control system topology and the authentication model. These authorization constraints can be found in table 6.2. A formal description of what it means that a permission can not be invoked and when a user is trapped can be found in 6.3.1 and 6.3.2.

Context-dependent A.C.	Description
Permission Uninvocable	a (user,scenario,permission) triple where the given user is granted the given permission during the given scenario which authorizes access to a security zone which can not be reached by the user in the given scenario
User Trapped	a (user,scenario,security zone) triple where the given user can reach the given security zone during the given scenario but becomes trapped and can not reach a security zone from which the user can leave the security system

Table 6.2: Context-Dependent Authorization Constraints

Definition 6.3.1 *Permission Invocability*
We say a for a given SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$, the permission p to access the security zone o granted to a user u during scenario S is not invocable if $(u, s, o) \in USO, (p, o) \in PO$ but $(u, s, p) \notin invocable$.
where

$$
\begin{aligned}
invocable &= \{(u, S, p) \in U \times \mathcal{S} \times P \mid \exists (u, S, o) \in accessible, (p, o) \in PO\} \\
&\cup \{(u, S, p) \in U \times \mathcal{S} \times P \mid \exists (u, S, o) \in accessible, \exists (o, o') \in \delta, (p, o') \in PO\} \\
accessible &= U \times \{(S, o) \in \mathcal{S} \times O \mid status(S, o, A) = unlocked,\ o \in I\} \\
&\cup \{(u, S, o) \in USO \mid status(S, o, A) \neq locked, o \in I\} \\
&\cup \{(u, S, o) \in U \times \mathcal{S} \times O \mid status(S, o, A) = unlocked, \\
&\quad \exists (o', o) \in \delta, (u, S, o') \in accessible\} \\
&\cup \{(u, S, o) \in USO \mid status(S, o, A) \neq locked, \\
&\quad \exists (o', o) \in \delta,\ (u, S, o') \in accessible\}
\end{aligned}
$$

with $\mathcal{S} = scenarios(C, \gamma)$

Definition 6.3.2 *User Trapped*
We say a for a given SACS model $\mathcal{P} = (\mathcal{M}, \mathcal{A}, \mathcal{V})$, a user u is trapped in a security zone o during scenario S if $(u, s, o) \in USO$ but $(u, s, o) \notin leavable$.
where

$$
\begin{aligned}
leavable &= \{(u, S, o) \in accessible \mid o \in I\} \\
&\cup \{(u, S, o) \in accessible \mid \exists (o, o') \in \delta, (u, S, o') \in leavable\}
\end{aligned}
$$

with $accessible$ as defined in 6.3.2.

Chapter 7

Implementation

In this chapter, we show how we constructed our prototype incremental policy verifier with VIATRA. First, we show how we translated the SACS formalism into an Ecore Model. Next, we will show how the access relation and the authentication statuses can be expressed as graph patterns on this model. This allows us to incrementally compute these relations as the match sets of graph patterns can be incrementally updated in VIATRA. We will also show how the various types of authorization constraints can be expressed as graph patterns. This allows us to incrementally verify them. Lastly, we will show how we incrementally compute the set of scenarios. The full source code of our implementation is available at `https://github.com/HansvdLaan/GRRBAC-Verifier/`

7.1 Ecore Model

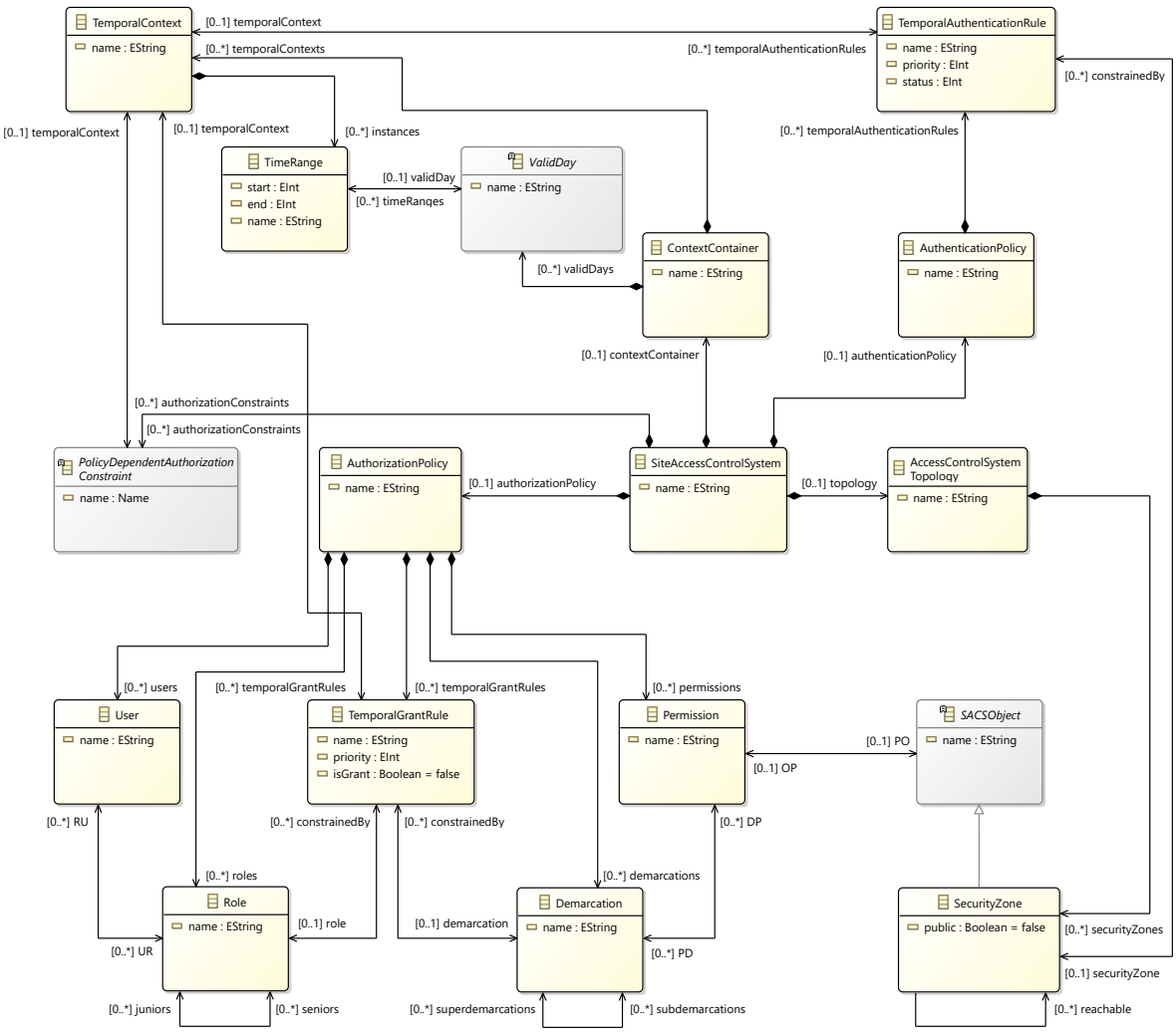


Figure 7.1: Ecore SACS Model Overview

The SACS formalism mainly deals with entities which are in relation with other entities. Therefore, the translation into Ecore was relatively straightforward. An overview of the Ecore SACS model can be found in Figure 7.1.

Four things are omitted from this overview. First, not all subclasses of the abstract authorization constraint class are shown. The policy-dependent authorization constraint subtypes (e.g. static/dynamic user-demarcation SoD) are implemented as separate subclasses. This allowed us to define separate patterns to find the violations of each constraint subtype. Secondly, not all subclasses of ValidDay are shown. This class is used to indicate what valid day of the week/month/year or day of the week/month combination the time interval of a time range occurs in. We again created separate subclasses for each of the possible types of valid day. Thirdly, we

omitted a class which is used to store intermediate results when computing the set of scenarios. Lastly, we omitted a class which is used for debugging purposes to work around a current limitation of VIATRA.

7.2 Policy Relations

We have defined 29 graph patterns to incrementally compute the various relations specified in Chapter 4. To give an impression of how the relations were formalized as VQL patterns on the SACS Ecore Model, we will show how we expressed the temporal grant, access and authentication statuses relations in VQL.

7.2.1 Temporal Grant Relation

To start, let us show the formalization of the temporal grant relation RSD into VQL. The VQL pattern definition resembles the set-based definition of this relation in Definition 4.5.13. The main difference is that we have to specify a special case to deal with the situations when there are no temporal grant rules which revoke a demarcation from a given role during a given scenario (i.e. $RevokePriority()$ has no matches and thus there is no maximum revoke priority). We have noticed that often our set-based definitions translated quite well into VQL.

This pattern makes use of quite a few other patterns. The $Scenario$ pattern computes all scenarios. $ScenarioTemporalContext$ finds which temporal contexts are a part of which scenarios. We will elaborate on these two patterns in Section 7.5.1. $connectedByTemporalGrantRule$ finds which roles are related via a temporal grant rule to a demarcation. $GrantPriority$ and $RevokePriority$ find the priorities of the temporal grant/revoke rules defined on the given role and demarcation which hold during the given scenario. $RevokePriority$ is defined similarly to $GrantPriority$.

```
1  pattern RSD(role: Role, scen: java Scenario, dem: Demarcation) {
2     find Scenarios(scen);
3     find connectedByTemporalGrantRule(role, dem); // perf. optm.
4     find GrantPriority(scen, role, dem, _); // perf. optm.
5     find RevokePriority(scen, role, dem, _); // perf. optm.
6     maxGrantPriority == max find GrantPriority(scen, role, dem, #p); [1]
7     maxRevokePriority == max find RevokePriority(scen, role, dem, #p2);
8     check(maxGrantPriority > maxRevokePriority);
9  } or { // when RevokePriority() has no matches
10    find Scenarios(scen);
```

[1] max is an aggregator. The # indicates which parameter of the called pattern should be aggregated (e.g with max the greatest number is selected)

```
11    find connectedByTemporalGrantRule(role, dem); // perf. optm.
12    find GrantPriority(scen, role, dem, _);
13    neg find RevokePriority(scen, role, dem, _);
14 }
```

```
1  pattern connectedByTemporalGrantRule(role: Role, dem: Demarcation) {
2     TemporalGrantRule.role(rule, role);
3     TemporalGrantRule.demarcation(rule, dem);
4  }
```

```
1  pattern GrantPriority(scen: java Scenario, role: Role, dem:
       Demarcation, priority: java Integer) {
2     find ScenarioTemporalContext(scen,context);
3     TemporalContext.temporalGrantRules(context, rule);
4     TemporalGrantRule.isGrant(rule, isGrant);
5     check(isGrant);
6     Role.constrainedBy(role, rule);
7     Demarcation.constrainedBy(dem, rule);
8     TemporalGrantRule.priority(rule, priority);
9  }
```

It is worth noting that the statements on line 3-5 and 11 in RSD are not strictly necessary. However, sometimes in VIATRA we can improve performance by adding extra constraints. In this case, by adding these constraints, we reduce the number of objects to consider in subsequent constraints (and thus increase performance).

7.2.2 Access Relation

In contrast to how we formally defined the access relation, we do not directly compute the access relation USO from the temporal grant relation RSD. Instead, we first compute the intermediary access relations USP, RSP and $RHSDH$ which are then used to compute the access relation USO.

```
1  pattern USO(user: User, scen: java Scenario, object:SACSObject) {
2     find USP(user, scen, perm);
3     Permission.PO(perm, object);
4  }
```

```
1  pattern USP(user: User, scen: java Scenario, perm:Permission) {
2     User.UR(user, role);
3     find RSP(role, scen, perm);
4  }
```

```
1  pattern RSP(role: Role, scen: java Scenario, perm:Permission) {
2     find RHSDH(role, scen, dem);
3     Demarcation.DP(dem, perm);
4  }
```

```
pattern RHSDH(role: Role, scen: java Scenario, demP:Demarcation) {
    Role.juniors*(role, roleP);
    find RSD(roleP, scen, dem);
    Demarcation.subdemarcations*(dem, demP);
}
```

We compute USP, RSP and $RHSDH$ separately as they are also needed when verifying the authorization constraints. In general, we tried to reuse computed (intermediary) relations as much as possible. This prevents that the match sets of equivalent subpatterns have to be (re)computed multiple times, increasing efficiency and reducing memory consumption.

7.2.3 Authentication Status

The pattern computing the temporal authentication status $status$ (for all security zones during all occurring scenarios) is defined similarly to the pattern computing the temporal grant relation RSD. A difference is that the locked, protected and unlocked statuses now have a default priority of -1. This removes the need to deal with the cases that for one or more types of actions no rule is defined covering that scenario separately.

The function $determineAuthenticationStatus()$ computes the authentication status given the found priorities. If no rules are defined for the given security zone during the given scenario, the default status is returned. This is the protected status. The locked status takes precedence over the protected status, which in turn takes precedence over that of the unlocked status.

```
pattern SecurityZoneAccessStatus(scen: java Scenario, zone:
    SecurityZone, status: java Integer) {
  find LockedPriority(scen, zone, _); // perf. optimization [2]
  maxLockedPriority == max find LockedPriority(scen, zone, #p1);
  maxProtectedPriority == max find ProtectedPriority(scen, zone, #p2);
  maxUnlockedPriority == max find UnlockedPriority(scen, zone, #p3);
  status == eval(MyQueryUtil.determineAuthenticationStatus(
   maxLockedPriority, maxProtectedPriority, maxUnlockedPriority));
}
```

[2] see https://www.eclipse.org/forums/index.php/t/1105611/ for a related discussion. (accessed on 22-1-2021)

7.3 Policy-dependent Authorization Constraints

The VQL patterns which verify the policy-dependent authorization constraints each look for constraint violations for a specific subtype of authorization constraint. To illustrate, the following VQL patterns try to find violations of user-permission SoD and static demarcation-permission cardinality constraints.

```
pattern SoDUPViolation(constraint: SoDUPConstraint, scen: java
    Scenario, user: User){
  SoDUPConstraint.left(constraint, left);
  SoDUPConstraint.right(constraint, right);
  AuthorizationConstraint.temporalContext(constraint, context);
  find ScenarioTemporalContext(scen, context);
  find USP(user, scen, left);
  find USP(user, scen, right);
}
```

```
pattern CardinalityDPViolation(constraint: CardinalityDPConstraint,
    usageCount: java Integer){
  CardinalityDPConstraint.permission(constraint, permission);
  CardinalityDPConstraint.bound(constraint, bound);
  usageCount == count Demarcation.DP(_, permission);
  check(usageCount > bound);
}
```

We implemented the following authorization constraint subtypes: user-role, user-demarcation, user-permission, role-demarcation, role-permission and demarcation-permission. As these kind of patterns are not that complex, we will not go into further detail about how we translated more policy-dependent authorization constraints subtypes into VQL.

7.4 Policy-independent Authorization Constraints

The patterns which detect the policy smells are also not that complex. The only exception being the patterns which check for the so-called God Users and the God Roles. VQL has no direct way of stating something should be in a relation with all elements of a certain type. We worked around this by detecting God Users as follows:

```
pattern GodUser(user: User) {
  currentSPCount == count find USP(user, _, _);
  permissionCount == count Permission(_);
  scenarioCount == count find Scenarios(_);
  maxSPCount == eval(permissionCount * scenarioCount);
  check(currentSPCount == maxSPCount);
```

```
}
```

One of the most time-consuming parts of the implementation proved to be the VQL patterns which try to find violations of the context-dependent authorization constraints. Although the $accessible$ and $leavable$ relations can be expressed rather directly as graph patterns, a considerable amount of time was required tweaking these patterns. Due to their recursive nature, their match sets have proven to be quite expensive to initialize and update. Furthermore, the underlying Rete networks could take quite a noticeable amount of memory. We experimented by adding extra constraints and factoring out subpatterns to enforce the underlying Rete network of the pattern to be structured more efficiently.

7.5 Computing Scenarios and Scenario Instances

We incrementally compute the set of scenarios $\mathcal{S}$ by computing a finite representation $T_{\mathcal{S}} \subseteq \mathcal{T}_{wm} \cup \mathcal{T}_y$ of the infinite set of scenario instances and inferring the set of scenarios from it.

7.5.1 Conceptual Idea

The idea behind our technique is that, because of how we construct $T_{\mathcal{S}}$, we claim that for all scenario instances one of the the following will occur, either: 1) there exists a time range $t_y \in T_{\mathcal{S}} \cap \mathcal{T}_y$ covering it and the scenario this instance belongs to is the same as the set of contexts covering t_y or 2) there does not exists such a t_y but there exists a $t_{wm} \in T_{\mathcal{S}} \cap \mathcal{T}_{wm}$ covering it and the scenario this instance belongs to is the same as the set of contexts covering t_{wm}. This means that for a given SACS model and given $T_{\mathcal{S}}$, the set of scenarios can also be defined as:

$$\begin{aligned} scenarios(C, \gamma) &= \{covers(t_y, C, \gamma) \mid t_y \in \mathcal{T}_{\mathcal{S}} \cap \mathcal{T}_y\} \\ &\cup \{covers(t_{wm}, C, \gamma) \mid t_{wm} \in T_{\mathcal{S}} \cap \mathcal{T}_{wm}, \\ &\quad \exists t \in \mathcal{T}_\tau, t \subseteq t_{wm}, \nexists t_y \in T_{\mathcal{S}} \cap \mathcal{T}_y, t \subseteq t_y\} \end{aligned}$$

This definition can be further simplified. We also claim that because of how we construct $T_{\mathcal{S}}$, if the set of all temporal context instances is not infinite, there will always exists a time instant covered for each $t_{wm} \in T_{\mathcal{S}}$, $t_{wm} \in \mathcal{T}_{wm}$ which is not covered by a $t_y \in T_{\mathcal{S}}$, $t_y \in \mathcal{T}_y$. As we will not deal with infinite sets of temporal context instances when analyzing realistic SACS models, this means that the set of

scenarios can also be defined as:

$$scenarios(C, \gamma) \quad = \quad \{covers(t, C, \gamma) \mid t \in T_{\mathcal{S}}\}$$

Since $T_{\mathcal{S}}$ is finite, the above definition is computable.

7.5.2 Overview

The process to compute $T_{\mathcal{S}}$ and the set of scenarios $\mathcal{S}$ conceptually consists of five steps:

1. First, we split the set of all temporal context instances T (not to be confused with $\mathcal{T}$, the set of all valid time ranges) into the subsets T_w, T_m, T_{wm}, T_y based on what each time range represents. T_w contains all time ranges which represent valid days of the week, T_m which represents valid days of the month, T_{wm} which represent valid days of the week/month combinations and T_y which represent valid days of the year.

   Please recall that we assume the temporal context $Always$ has a finite set of temporal context instances and is defined a set of time ranges representing on the level of days of the week (($[0, 1439], Mon, \_, \_, \_$), ($[0, 1439], Tue, \_, \_, \_$), etc.).

2. From each subset, we compute the set of disjoint time ranges T'_w, T'_m, T'_{wm}, T'_y which partition the time ranges of the original subsets into maximal disjoint time ranges such that for each disjoint time range, any two moments in time covered by it overlap with the same time ranges from the original subset (T_w, T_m, T_{wm} and T_y). An example partitioning is shown in Figure 7.2.

3. Next, we combine T'_w, T'_m and T'_{wm} and compute another set of maximal disjoint time ranges T''_{wm} which partition the combined sets. Similarly, we combine T''_{wm} and T'_y for all days of year on which time intervals are specified and compute the set of maximal disjoint time ranges T''_y.

4. Afterwards, we combine T''_{wm} and T''_y into $T_{\mathcal{S}}$. We claim that because of how we constructed these sets that for each time range t contained in T''_y we know all time instants covered by t overlap with the same temporal context instances and will belong to the same scenario. Furthermore, we claim that for each time range t contained in T''_{wm} all time instants covered by t which are not covered by any time range contained in T''_y will belong to the same scenario.

5. During all previous steps, we keep of temporal context instances overlap with which the scenario instances. Given this knowledge and C, γ and $T_{\mathcal{S}}$ we can compute the set of scenarios $\mathcal{S}$.

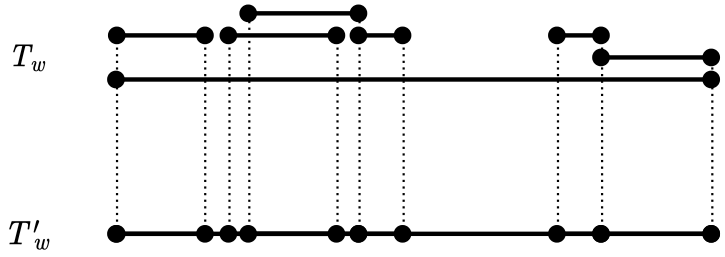


Figure 7.2: Example partitioning of T_w into T'_w for an arbitrary day of the week

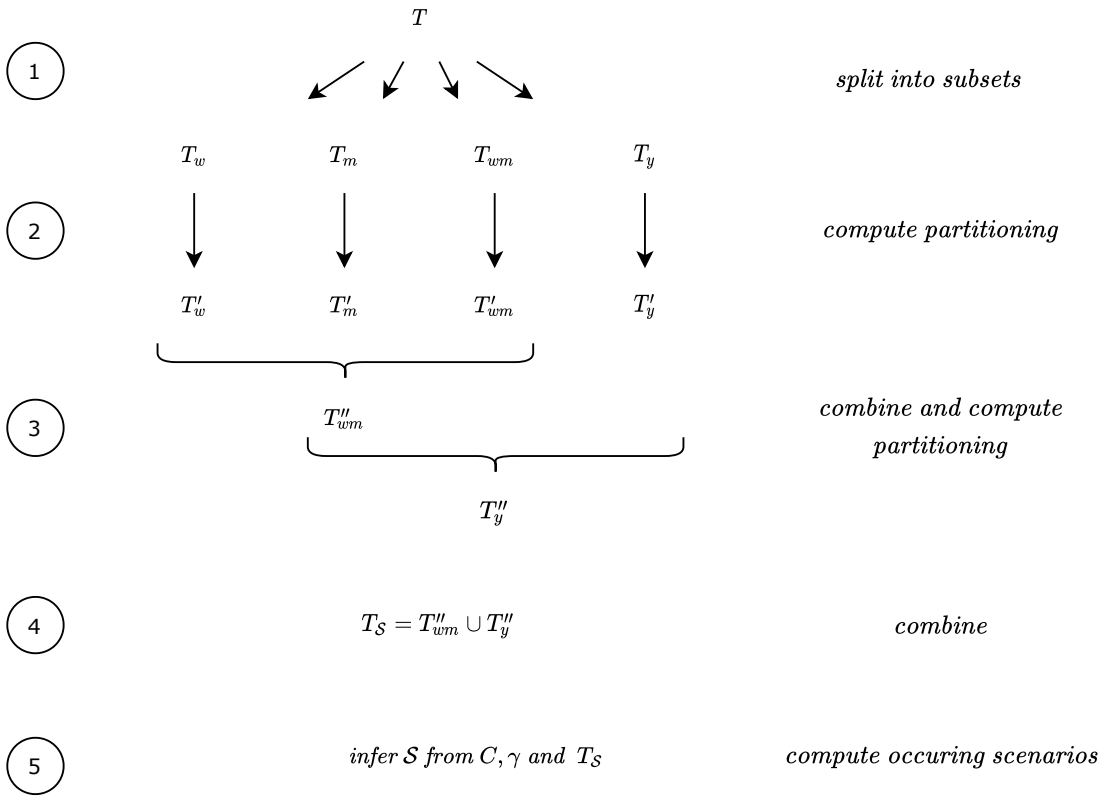


Figure 7.3: steps to compute $\mathcal{S}$ and $T_{\mathcal{S}}$

An overview of all steps can be found in Figure 7.3. An example of those steps performed on a slightly modified version of ACME's access control policy's set of temporal contexts (see Table 4.1) is shown in Figure 7.4. Every step except the second is done in a declarative manner with graph patterns expressed in VQL (and thus they are thus automatically performed incrementally thanks to the VIATRA framework).

We have developed algorithms to incrementally compute T'_w, T'_m, T'_{wm}, T'_y as this was non-trivial to perform declaratively with graph patterns while also keeping track of the relation between temporal context instances and their partitioning. Naively, we would need for any given day with n associated temporal context instances store

at least $n \times (n-1)$ comparisons. The algorithms are integrated with VIATRA as a event-based model transformation. Whenever a temporal context instance is added or removed, VIATRA runs the appropriate algorithm.

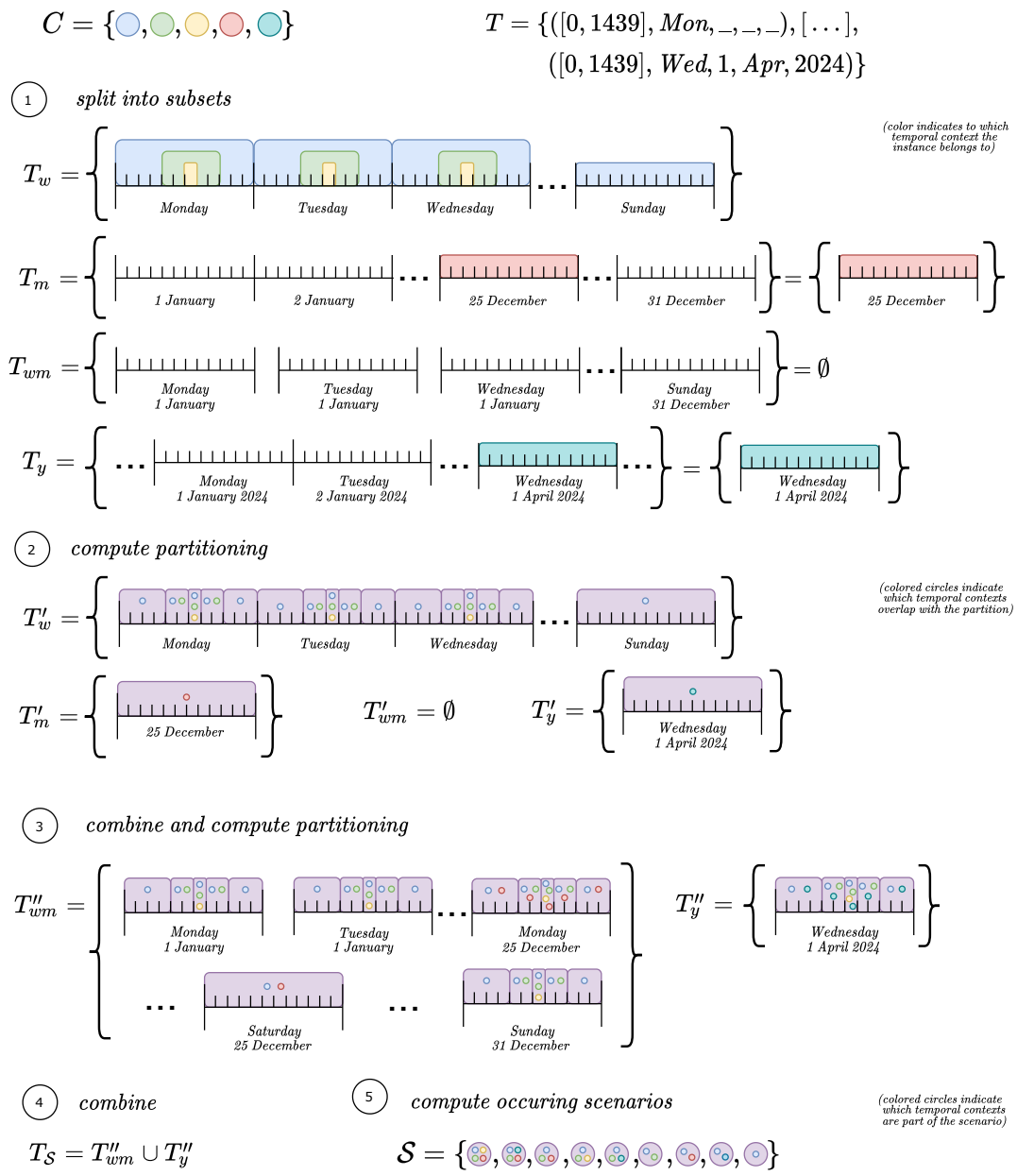


Figure 7.4: steps to compute $\mathcal{S}$ and $T_{\mathcal{S}}$ example

In total, we established 16 VQL patterns to compute $\mathcal{S}$. We refer the reader to our public code repository to see the implementation of all steps. To facilitate the implementation, we used a different definition of $Always$ in our implementation compared to how we defined it formally. Namely, we assumed that $Always$ was defined as the set of time ranges covering the whole day for every day of the day week, day

of the month and day of the week/month combination. Furthermore, for every day of the year on which another temporal context instance had been defined, we added a time range covering that whole day to $Always$ during runtime. Practically, this means that in our implementation, the $Always$ temporal context has at least 2935 temporal context instances instead of 7.

In the rest of this chapter, we will focus (1) at the algorithm we have developed to incrementally compute T'_w, T'_m, T'_{wm}, T'_y and (2) the VQL patterns which infer $\mathcal{S}$ from C, γ and $T_{\mathcal{S}}$. We want to highlight (1) as the developed algorithms are non-trivial to understand from our implementation only. We will look at (2) as our evaluation has shown the implementation of this step is inefficient and we believe other VIATRA practitioners might learn from it.

7.5.3 Incremental Computation of Initial Partitions

That the computation of the partitions is done incrementally means that, whenever a temporal context instance is added to or removed from T'_w, T'_m, T'_{wm} and T'_y, the partition will be updated accordingly instead of being recomputed from scratch. We keep track of a relation which maps the temporal context instances to the scenario instances they overlap with. The algorithms which perform this incremental computation are shown in Algorithm 1 and Algorithm 2.

The intuition behind the algorithms is that each day of the week/month/year has an interval from 0 up to and including 1439 associated with it which is incrementally partitioned based on the intervals of the temporal context instances specified on that day. When a temporal context instance is added, the partitioning is either refined or stays the same. Whenever a temporal context instance is removed, the partitioning is either coarsened or stays the same. The partitioning algorithms also keep track of a relation $overlap$ which maps temporal context instances to their partitions.

The algorithm which processes the newly added temporal context instances first finds all existing partitions which overlap with the new temporal context instance. This can be achieved with the help of an interval tree. An interval tree is a tree data structure which holds intervals and can efficiently find all intervals that overlap with a given interval. Then, for each overlapping partition, it is either kept further subdivided into two or three subpartitions if none, one or both of the bounds of the newly added temporal context instance fall between the bounds (excluding the bounds themselves) of the scenario instances with $overlap$ being updated accordingly.

The complexity to find all intervals that overlap with a given interval depends on the type of interval tree. In our implementation, we use an augmented interval tree. With this type of interval tree, adding intervals to the tree, removing intervals from the tree and finding all intervals that overlap with a given interval take $O(\log n)$ time, with n being the number of intervals in the tree.

The algorithm which processes removed temporal context instances also starts by finding all partitions in the same manner as the other algorithm. Then, for each found overlapping partition, it updates $overlap$. Lastly, it checks if the partition with the lowest lower bound of the overlaps can be merged with the partition before it (if it exists) and if the partition with the highest upper bound can be merged with the partition after it (again if it exists).

Algorithm 1: Process Addition of Temporal Context Instance

Input: $t_{new} = [\underline{t_{new}}, \overline{t_{new}}]$, P = *partitions associated with the same day as* t_{new}

$P_{overlap} \leftarrow \{p \mid p \in P,\ t_{new}\ overlaps\ with\ p\ \}$
if $|P_{overlap}| = 1$ **then**
  $p = [\underline{p}, \overline{p}] \leftarrow only\ element\ of\ P_{overlap}$
  if $t_{new} = p$ **then** // case 1
    $overlap \leftarrow overlap \cup \{(t_{new}, p)\}$
  else if $\underline{t_{new}} > \underline{p}$ *and* $\overline{t_{new}} < \overline{p}$ **then** // case 2
    $p_{new1} \leftarrow [\underline{t_{new}}, \overline{t_{new}}]$
    $p_{new2} \leftarrow [\overline{t_{new}} + 1, \overline{p}]$
    $resize\ p\ to\ [\underline{p}, \underline{t_{new}} - 1]$
    $overlap \leftarrow overlap \cup \{(t, p_{new1}) \mid (t, p) \in overlap\}$
    $overlap \leftarrow overlap \cup \{(t, p_{new2}) \mid (t, p) \in overlap\}$
    $overlap \leftarrow overlap \cup \{(t_{new}, p_{new1})\}$
  else if $\overline{t_{new}} < \overline{p}$ **then** // case 3
    $p_{new} \leftarrow [\underline{t_{new}}, \overline{t_{new}}]$
    $resize\ p\ to\ [\overline{t_{new}} + 1, \overline{p}]$
    $overlap \leftarrow overlap \cup \{(t, p_{new}) \mid (t, p) \in overlap\}$
    $overlap \leftarrow overlap \cup \{(t_{new}, p_{new})\}$
  else if $\underline{t_{new}} > \underline{p}$ **then** // case 4
    $p_{new} \leftarrow [\underline{t_{new}}, \overline{t_{new}}]$
    $resize\ p\ to\ [\underline{t_{new}}, \underline{t_{new}} - 1]$
    $overlap \leftarrow overlap \cup \{(t, p_{new}) \mid (t, p) \in overlap\}$
    $overlap \leftarrow overlap \cup \{(t_{new}, p_{new})\}$
else
  foreach $p = [\underline{p}, \overline{p}] \in P_{overlap}$ **do**
    if $\overline{t_{new}} < \overline{p}$ **then** // case 5
      // same as with $|p_{overlap}| = 1$, case 3
    else if $\underline{t_{new}} > \underline{p}$ **then** // case 6
      // same as with $|p_{overlap}| = 1$, case 4
    else // case 7
      $overlap \leftarrow overlap \cup \{(t_{new}, p)\}$

Algorithm 2: Process Removal Of Temporal Context Instance

Input: $t_{rem} = [\underline{t_{new}}, \overline{t_{new}}], P = partitions\ associated\ with\ the\ same\ day\ as\ t_{rem}$

$P_{overlap} \leftarrow \{p \mid p \in P,\ t_{rem}\ overlaps\ with\ p\ \}$

$p_{first} \leftarrow interval\ with\ lowest\ lowerbound\ of\ P_{overlap}$

$p_{last} \leftarrow interval\ with\ highest\ upperbound\ of\ P_{overlap}$

$overlap \leftarrow overlap \cap \{(t_{rem}, p) \mid (t_{rem}, p) \in overlap\}$

if $\exists p = [\underline{p}, \underline{p_{first}} - 1] \in P \wedge overlap^{-1}(p) = overlap^{-1}(p_{first})$ **then**

  $merge\ p\ and\ p_{first}$

if $\exists p = [\overline{p_{last}} + 1, \overline{p}] \in P \wedge overlap^{-1}(p) = overlap^{-1}(p_{last})$ **then**

  $merge\ p\ and\ p_{last}$

7.5.4 Incremental Computation of Occuring Scenarios

We infer $\mathcal{S}$ from C, γ and $T_{\mathcal{S}}$ with the help of four VQL patterns. We have renamed some patterns and simplified calls to other patterns to facilitate their explanation.

The pattern $TWMPP\_TimeRange\_To\_Scenario$ computes for all time ranges $t_{wm} \in T''_{wm}$ the set of temporal contexts covering t_{wm}. $TYPP\_TimeRange\_To\_Scenario$ computes the same for all time ranges $t_y \in T''_y$. They are defined similarly. As we discussed previously, for any given time range in $t \in T_{\mathcal{S}}$, the set of all temporal contexts covering t corresponds to an occurring scenario. Thus, these patterns find occurring scenarios. We group the found temporal contexts covering a given time range into a scenario with the help of a custom aggregator $distinct$. Please note that scenarios are just renamed sets of temporal contexts (i.e. `Set<TemporalContext>`)[3].

$Scenarios$ aggregates the scenarios found by $TWMPP\_TimeRange\_To\_Scenario$ and $TYPP\_TimeRange\_To\_Scenario$. This corresponds to the set of all occurring scenarios. $ScenarioTemporalContext$ states which temporal contexts are part of which scenarios as a pattern.

```
1 pattern TWMPP_TimeRange_To_Scenario(validDay: ValidDay, starttime:
    java Integer, endtime: java Integer, scen: java Scenario) {
2   ValidDayOfWeekMonth(validDay); // perf. optimization
3   find TWMPP_TimeRange(validDay, starttime, endtime); // Simplified.
    Matches any t_wm ∈ T''_wm
```

[3]This was done since as a workaround since VIATRA at the moment does not fully support classes with generics, see: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=564426` (accessed on 22-1-2021)

```
  scen == distinct find TWMPP_TimeRange_To_TemporalContext(validDay,
    starttime, endtime, #context); // Custom aggregator. Groups all
    distinct temporal contexts into a scenario, i.e. a set of temporal
     contexts.
}
```

```
pattern Scenarios(scen: java Scenario) {
  find TWMPP_TimeRange_To_Scenario(_, _, _, scen);
} or {
  find TYPP_TimeRange_To_Scenario(_, _, _, scen);
}
```

```
pattern ScenarioTemporalContext(scen: java Scenario, context:
    TemporalContext) {
  find TWMPP_TimeRange_To_Scenario(day, start, end, scen);
  find TWMPP_TimeRange_To_TemporalContext(day, start, end, context);
} or {
  find TYPP_TimeRange_To_Scenario(day, start, end, scen);
  find TYPP_TimeRange_To_TemporalContext(day, start, end, context);
}
```

Our current implementation has two problems, which are interesting to note. The first problem is that given how we compute scenarios our prototype unnecessarily computes various relations and proves authorization constraints for spurious scenarios. The root cause of this problem is that the Rete engine always directly propagates new and disappeared matches. It does not let the match set of $Scenarios$ stabilize beforehand. While the set of scenarios is being (re)computed, spurious scenarios are found. These correspond to intermediary results. Since all newly found scenarios will directly be propagated to all other patterns, this means that time will be spent unnecessarily computing and proving properties about these intermediate results.

The second problem is that much work has to be redone when temporal contexts are removed. When a temporal context is removed, first all scenarios matches given this temporal context disappear, and then new scenarios matches without this temporal context appear. For these new scenarios matches, many relations have to be computed anew, and authorization constraints have to be reverified. This is not efficient; it would be better if our prototype knew the scenarios were slightly modified and more computations could be reused.

Chapter 8

Evaluation

In this chapter, we evaluate and discuss our prototypical incremental policy verifier based on an actual access control policy provided by Nedap. We will benchmark the time/memory required to initialize the engine and the time required to reverify the model after many types of authorization/authentication model modifications and when adding new authorization constraints. We aim to get an impression of:

- the impact of incrementalizing the verification,
- to what degree the type and nature of a change has an impact on performance,
- the cost of verifying context-dependent authorization constraints,
- and the scalability of our prototype.

We use these results to draw conclusions about our prototype and our approach. The source code of our verification engine, the anonymized access control policy and all test cases are publicly available at `https://github.com/HansvdLaan/GRRBAC-Verifier`. For demonstration purposes, we also developed a CLI version of the verification engine.

8.1 Business Case

We were graciously provided with an actual physical access control system's authorization model, authentication model and building topology by Nedap. According to their security experts, the system is representative of physical access control systems of small-medium enterprises. For privacy and security reasons, we can not disclose the name of the company who's access control system we were provided with and their access control policy, published alongside this research, has been fully anonymized.

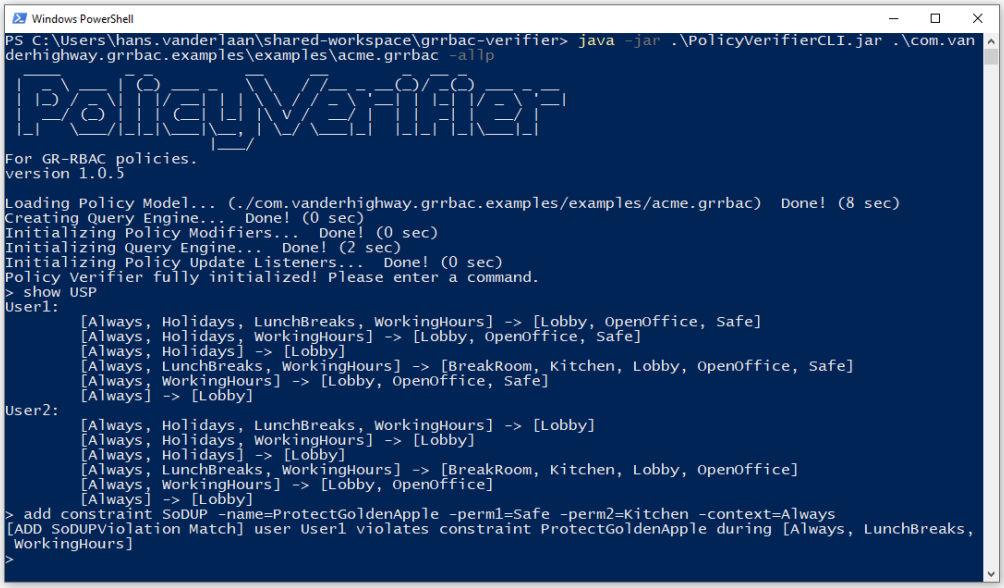


Figure 8.1: Policy Verifier CLI

8.1.1 Policy Conversion

We have converted the given policy into our newly-defined SACS format. The conversion of the provided authorization model into GR-RBAC was less than straightforward and we required the assistance from security experts in the company.

Explaining the complete format of the provided authorization model is out of the scope for this research. However, in short, it consists of Users, Templates, Entrance Groups and Entrances. Entrance groups are sets of entrances. Entrances are access controlled doors. Users can be assigned templates and can be granted time-dependent access to entrance groups and entrances. Templates are granted time-dependent access to entrance groups and entrances. There is no construct akin to GR-RBACS's role/demarcation hierarchies.

We transform templates into roles and entrance groups into demarcations. When an entrance group is granted directly to a user instead of via a template, we create a new intermediary template which is given the entrance group and assigned to the user. This substitutes the direct assignment. We employ a similar technique when entrances are directly assigned to users or templates. We call the roles/demarcations which are the result of this substitution *proxy roles/demarcations*.

Entrance groups and entrances are often granted directly to users. To prevent an unrealistic explosion in the number of roles and demarcations in the translated model,

we group a user's proxy roles and proxy demarcations together whenever possible. We also remove duplicates between users.

The translation of the authentication model and the building topology in our format was less challenging. From a practical standpoint, our formalisms can be seen as simplifications of those used by Nedap. However, there is a major difference that currently, Nedap provides access on the level of doors while SACS defines access on the level of zones. To deal with this discrepancy, we model doors and rooms as zones.

We model rooms as security zones which are always unlocked. Doors are modelled as two separate security zones, one for each door side. We model each door side separately as they can have different authentication status for a given point in time (e.g., one side of the door is protected with access control while the other side is unlocked). If a door side is always locked, we do not model this side. If the door side has no installed access control measures, we remove the newly created security zone and connect the security zones representing the adjacent rooms directly. We model the perimeter of the building as a security zone.

Unfortunately, the provided authentication model and building topology were incomplete. There are 78 entrances in the authorization model which are missing in the authentication model and the building topology. However, all doors which were central to the building are present.

8.1.2 Authorization Constraints

The provided policy has no explicitly defined authorization constraints. However, we found 15 implicit authorization constraints. These were added to the policy and thus will be reverified after each policy change. They were found through discussions with facility management of the company whose access control policy we were provided and with security experts from Nedap. Specifically, we have found 13 user-role SoD constraints and 2 user-role prerequisite constraints.

Interestingly, we have found no BoD or cardinality constraints nor constraints which are not on the user-role level. We suspect because in the original policy, what have now become demarcations often represented groups of entrances which were grouped because the entrances were considered to be part of the same room. Thus, they were not designed with implicit constraints in mind.

8.1.3 Model Statistics

The translated authorization model has 237 (unique) users, 164 roles (of which 141 proxy roles), 93 demarcations (of which 52 proxy demarcations), 160 permissions and 613 temporal grant rules. Regarding the (computed) relations, we have: $|UR| = 674$, $|DP| = 530$, $|USD| = 181160$, $|USP| = 490489$, $|RSD| = 22934$ and $|RSP| = 64587$. The authentication model consists of 450 temporal authentication rules. The translated building topology consists of 420 security zones of which 260 representing rooms and 160 representing access-controlled door sides. An overview of the building topology can be found in Figure 8.2. There are 37 policy-specific temporal contexts with a total 282 of temporal context instances. Interestingly, all instances are defined on a day of the week basis. The "Always" temporal context has 2935 temporal context instances. Furthermore, 478 temporal grant rules and 260 temporal authentication rules have as context "Always". This latter was to be expected, as this equals the number of rooms (which are always unlocked). In total, the 38 temporal contexts give rise to 40 scenarios with a total of 54534 scenario instances.

Regarding the authorization constraints and policy smells; we have found 6 authorization constraint violations and found 10062 context-dependent authorization constraint violations. This contains 9590 cases in which a user can get trapped in a security zone during a scenario and 472 cases in which a user can not invoke a granted permission during a scenario. Due to the missing doors, it is not possible to know for sure if the found context-dependent authorization constraints violations are spurious or genuine. For every violation, there could be a path which is not present in our conversion due to the missing doors.

8.2 Benchmark Execution

8.2.1 Experimental Setup

We focused on evaluating the following aspects of our prototype:

Impact of Modification Type & Nature

We have created various collections of test cases for most authorization and authentication model modifications. We also made test cases which add new authorization constraints. This allows us to get an impression to what degree the type of a change influence the time it takes to reverify the model. We made each test case distinct to get a similar impression about the impact of the nature of the change.

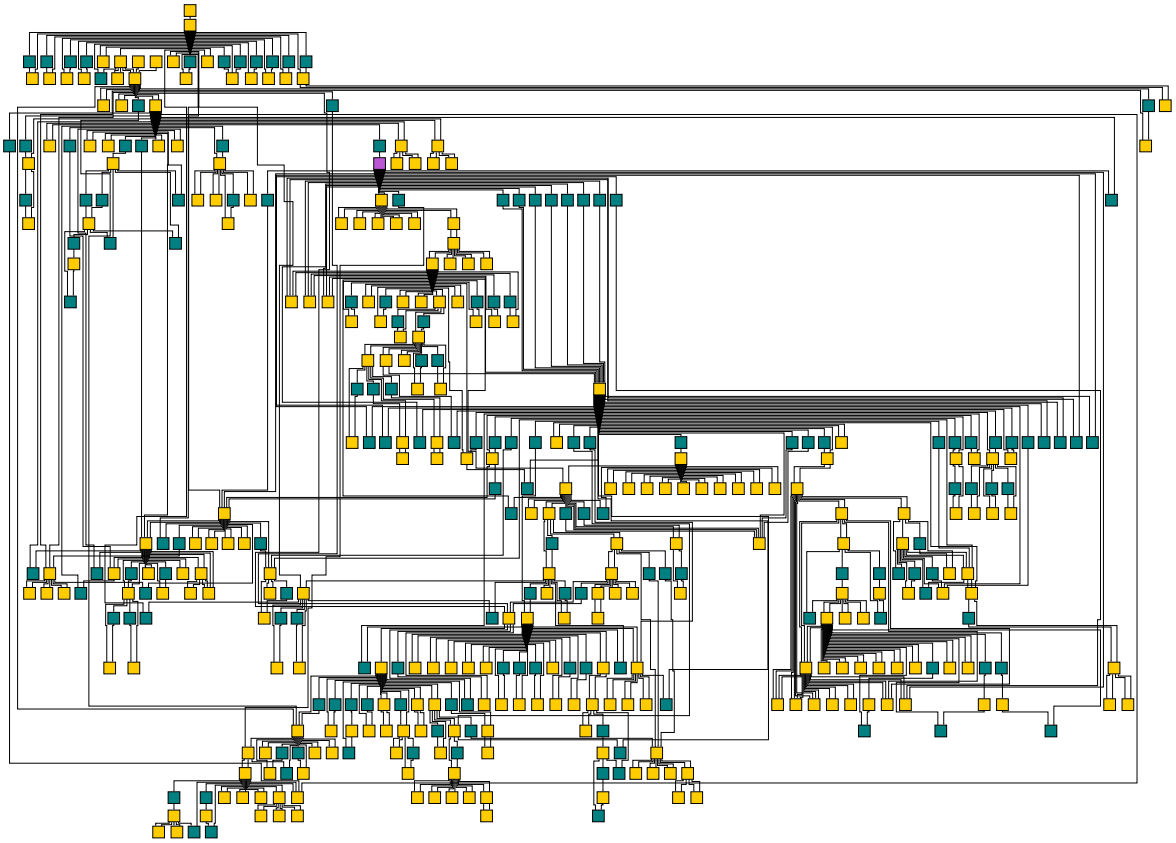


Figure 8.2: Building Topology

■ : Room ■ : Doorside ■ : Perimeter

→ : Reachability

Batch vs Incremental

To get an impression of the achieved speed-up by incrementalizing the verification, we evaluate the difference between caching and no caching, i.e. incremental and batch verification. We use the incremental Rete engine as a stand-in batch engine. Research has shown that it also works efficiently as a batch engine [70] (sometimes being faster than the local search engine for model sizes comparable to ours [70]). We take the initialization time of the incremental engine as the time required to (re)verify all authorization constraints (including the policy smells) in batch.

We initially aimed to use VIATRA's non-incremental local search engine as the batch engine. However, given how we have implemented our prototype, this engine is unreasonably inefficient for the verification task at hand. Since the set of scenarios is computed through graph-patterns and this engine performs no caching, the set of scenarios has to be recomputed from scratch when checking each type of authorization constraint. The same holds for e.g. the temporal grant and temporal authentication relations. Practically, we observed that checking one type of authorization constraint with the non-incremental local search engine took more time than checking the whole model with the incremental Rete engine. Thus, we decided to use the Rete engine as a stand-in batch engine.

Cost of Verifying Context-Dependent Authorization Constraints

We measure the impact of omitting context-dependent authorization constraints and all the patterns required to verify them to get an impression of how expensive constraints of this type are to verify. As we explained in Section 7.4 they are verified with a recursive graph pattern which determines accessibility rather naively. Computing accessibility this way appears to be quite expensive. This holds both for initializing and updating the set of accessible security zones.

Scalability

To assess the scalability of our solution, we have created scaled-up versions of our business case in which all authorization, authentication and topology model entities or all scenarios are duplicated 1, 2, 3 and 4 times.

8.2.2 Measurement Strategies

To guarantee the measurements are independent, each test case is run individually. This is achieved by simulating that the model is reloaded and the engine is reinitial-

ized for every test case. This is achieved with the help of CRIU.

CRIU [1] can freeze a running application, checkpoint its state to a collection of files, and restore the application from the point it was frozen at. We initialize the engine once for each collection of test cases. When this is done, the initialized engine is frozen and saved. The initialized engine is then restored for each individual test case.

Measurement of Execution Time

If a test case is completed within a timeout of 5 minutes, the measurement is considered successful, and the measurement results are saved. We will compute the average and maximum time. On a timeout, the measurement is terminated, discarded, and a timeout is reported. For each test, we will present the percentage of test cases which timed out. Timeouts can occur when it takes a lot of effort to compute the impact of the model change.

We do not disregard all measurements of a collection of test cases upon one or more time outs as we have observed a noticeable variation in the time required to reverify the model based on the nature of the change. However, when inspecting the results, it is thus important to take the number of time outs into account.

For all test cases on the default model, the median value of 3 runs was taken. For the test cases executed on scaled-up models, only one run was performed as each run takes a significant amount of time. For a few types of model changes, we were also forced to decrease the number of test cases of that model change during the scalability test.

We will use the simplification that the initialization time of the engine will roughly correspond to the time which will be required to reinitialize the engine after the model change. There are cases for which this does not hold. In the case a model entity is removed which has a lot of relations, the actual reinitialization time might lower. However, from a practical standpoint it is not feasible to measure the reinitialization time of the engine after each model modification as this can take a considerable amount of time.

[1] `https://criu.org/Main_Page`

Measurement of Memory Consumption

The measurement of memory consumption was done as in [70]. In a binary search-like fashion, the available memory to the JVM is decreased in it is observed if the allotted memory is still enough to initialize the engine. With 9 tests and an initial limit of 6.4GB, this allows us to measure the peak memory consumption with a precision of 25 MB.

Please note that the measurements for memory consumption and execution time were performed separately. Low memory can force the JVM to perform more garbage collection, decreasing performance. For measurement of execution times, the JVM was allowed a maximum of 8GB of memory.

8.2.3 Environment

The benchmark was performed on two HP ProLiant DL380 G7 servers with each two six-core Intel(R) Xeon(R) CPU X5660 @ 2.80GHz with 96 GB of RAM each (12x 8192MB DDR3 at 1333 MHz). The servers were running Debian (64-bit) with a backported Linux 5.4.0-0 of the Linux kernel with hyperthreading turned off and the OpenJDK 11.9.0.1 runtime. We backported an older version of the Linux kernel since newer versions contained a bug in which caused JVMs to be not restored properly with CRIU[2]. We used VIATRA version 2.5.0.

[2]Since the time of writing, this bug has been fixed, see: `https://lkml.org/lkml/2020/10/15/582` (accessed on 22-1-2021)

8.2.4 Results

Initialization Time/Memory The time and memory required to initialize the VIATRA query engine is measured. We performed 8 measurements for the initialization time. We disregard the time it takes to load the XML file of the GR-RBAC model.

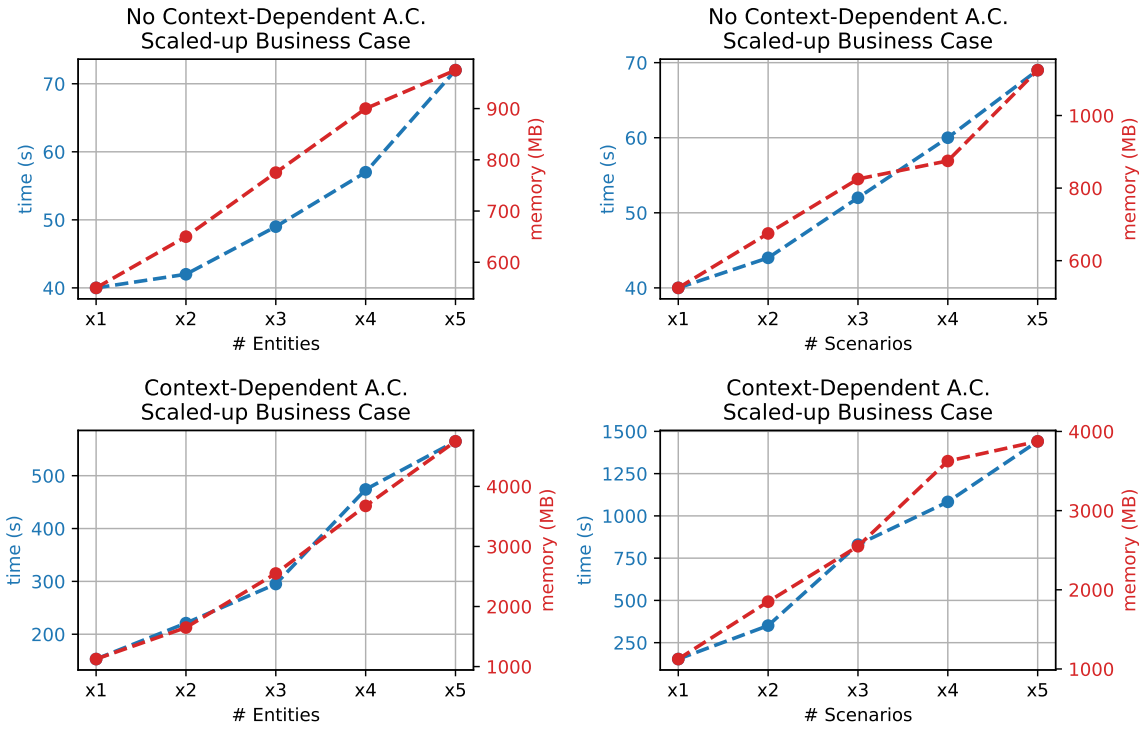


Figure 8.3: Initialization Time/Memory Consumption Comparison

Please recall that we consider the time required to initialize the engine as the time required to (re)verify all authorization constraints in batch. Thus, we consider that it takes 40 seconds to verify all authorization constraints on the business case in batch when context-dependent authorization constraints are not taken into account and 153 seconds when they are.

Add Entities We tried to get an impression of the time required to reverify the model upon the creation of a new user (1), role (2), demarcation (3) or permission (4). Upon the creation of a new permission, an unconnected non-public security zone is also created since permissions should always be associated with a zone. For (1)-(4), we generated 80 test cases.

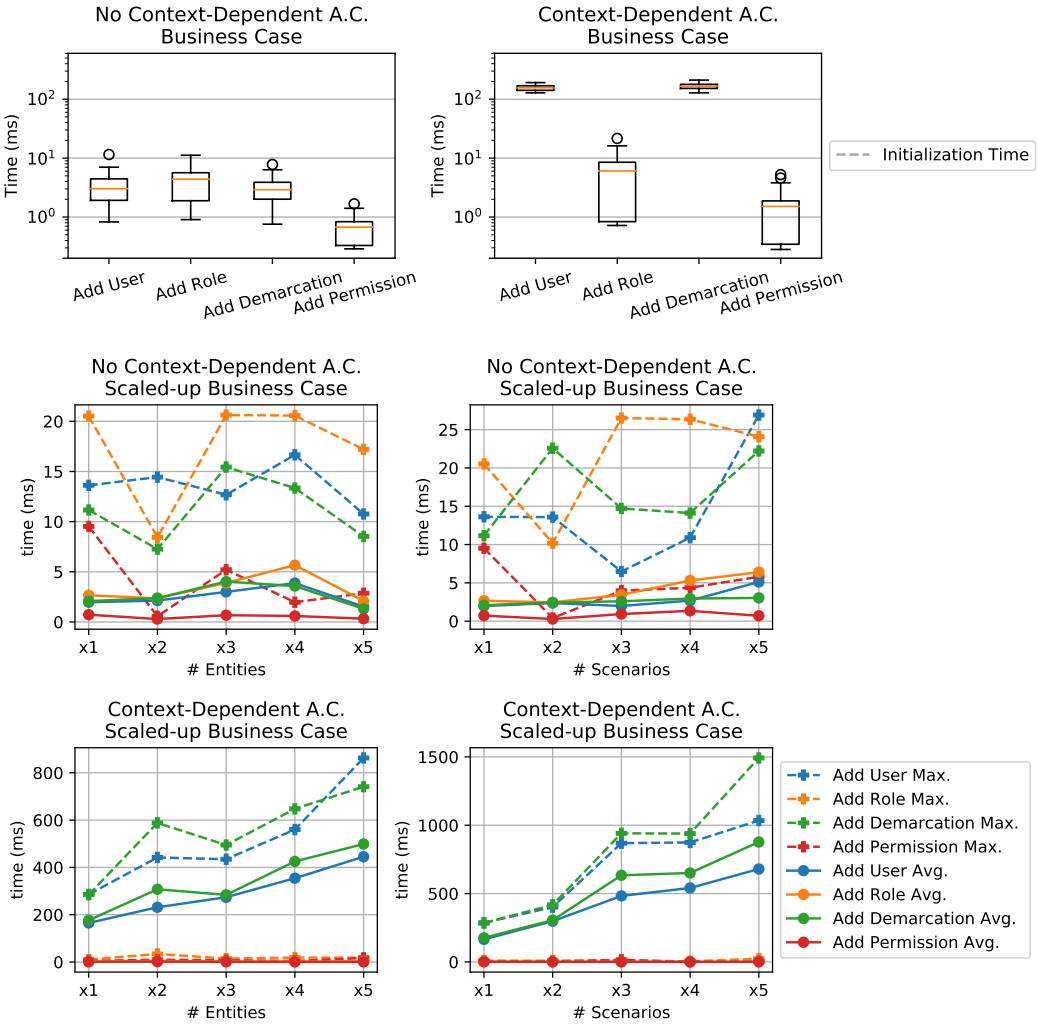


Figure 8.4: Add Entities Measurement Results

Remove Entities We measured the time required to reverify the model after a user (1), role (2), demarcation (3) or permission (4) is removed. For (1) and (4), we tried removing each user and permission individually. For (2) and (3), we tried removing each non-proxy role/demarcation (i.e. those which are not mapped to templates/entrance groups which came into existence to facilitate the translation) individually. For (1)-(4), we respectively had 237, 23, 41 and 160 different test cases.

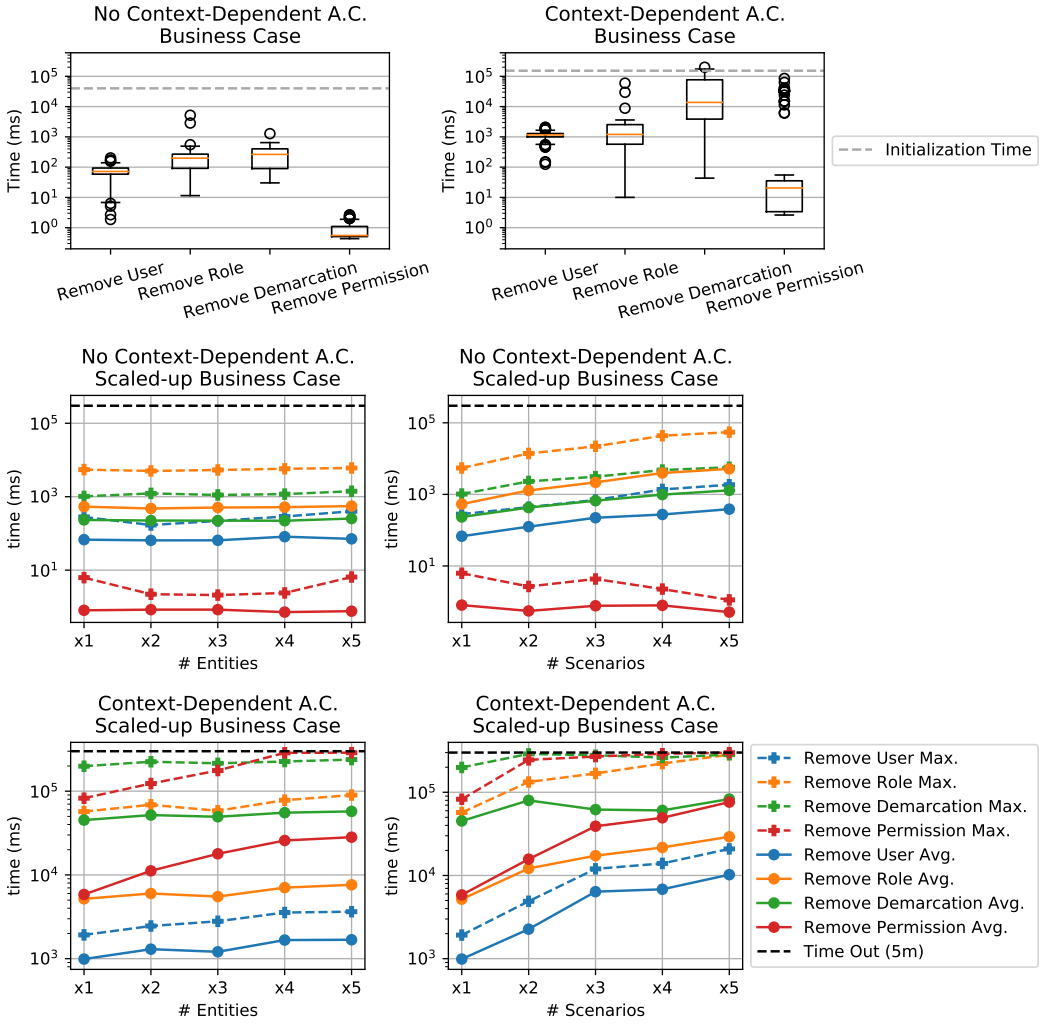


Figure 8.5: Remove Entities Measurement Results

The following percentage of test cases timed out when taking Context-Dependent A.C. into account:

Modification Type \ # Entities	1x	2x	3x	4x	5x
Remove Demarcation	7.3% (3)	7.3% (3)	7.3% (3)	7.3% (3)	7.3% (3)
Remove Permission	-	-	-	-	3.8% (6)

Table 8.1: Timeouts of Remove Entities Test Cases with Context-Dependent A.C. and Increasing Number of Model Entities

Modification Type \ # Scen.	1x	2x	3x	4x	5x
Remove User	-	-	-	0.4% (1)	-
Remove Role	-	-	-	8.7% (2)	8.7% (2)
Remove Demarcation	7.3% (3)	17.1% (7)	39% (16)	43.9% (18)	46.3% (19)
Remove Permission	-	-	0.6% (1)	3.1% (5)	6.9% (11)

Table 8.2: Timeouts of Remove Entities Test Cases with Context-Dependent A.C. and Increasing Number of Scenarios

Assign Entities We measured the time required to reverify the model after the assignment a role to a user (1) and the assignment of a permission to a demarcation (2). For (1), for each existing user, we tried assigning (up to 10) new non-proxy roles individually. For (2), for each existing non-proxy demarcation, we tried assigning (up to 10) new permissions individually. For (1) and (2), we respectively had 2370 and 410 different test cases.

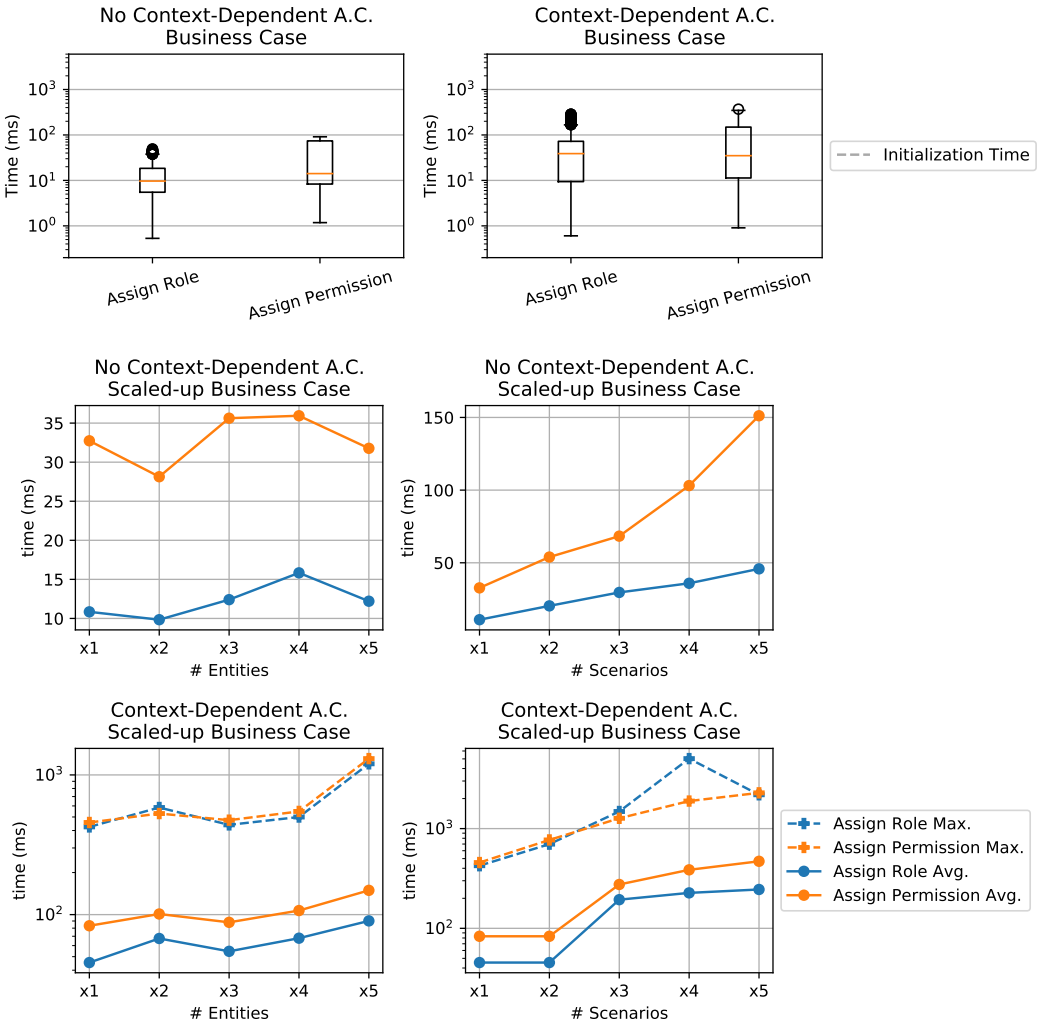


Figure 8.6: Assign Entities Measurement Results

Deassign Entities We measured the time required to reverify the model after the deassignment of a role from a user (1) and the deassignment of a permission from a demarcation (2). For (1), for each existing user, we tried deassigning each non-proxy role individually. For (2), for each existing non-proxy demarcation, we tried deassigning (up to 10) permission individually. For (1) and (2), we respectively had 486 and 103 different test cases.

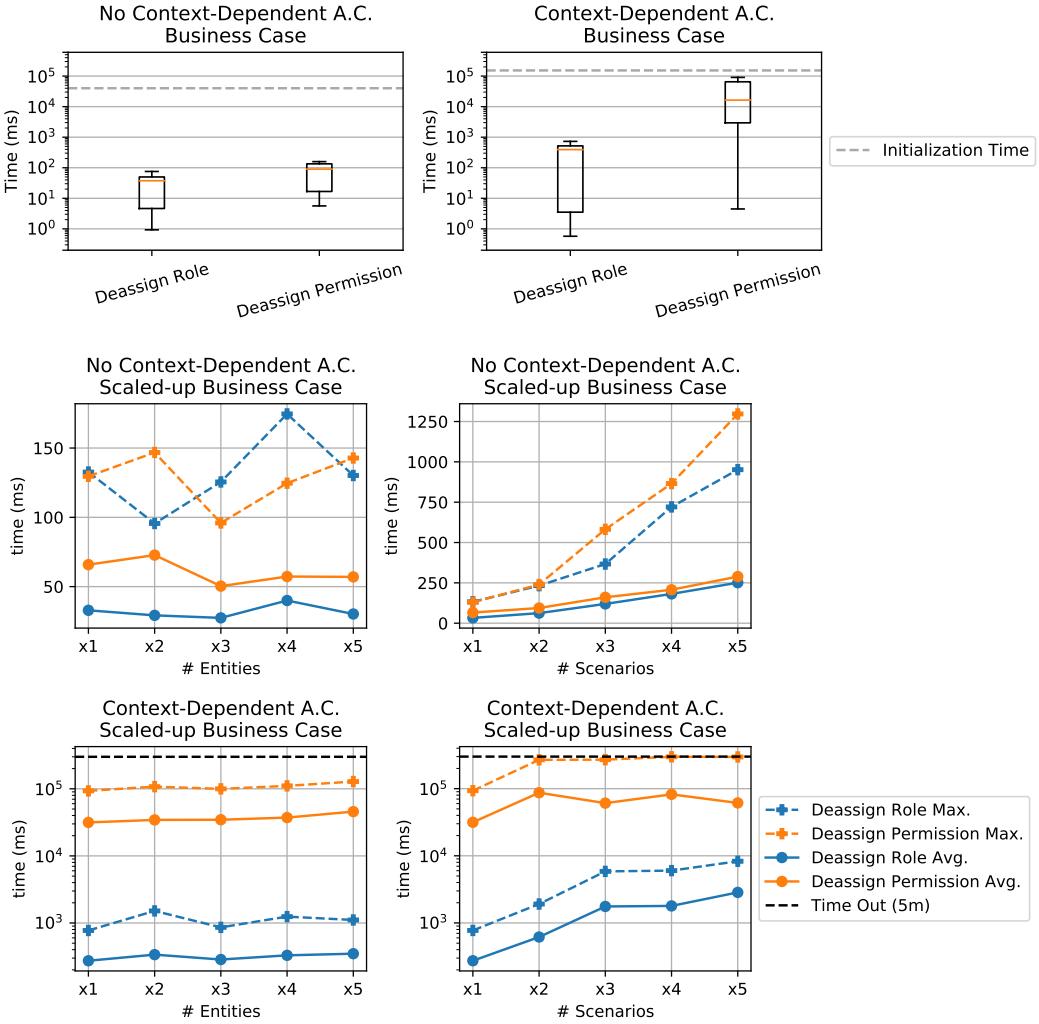


Figure 8.7: Deassign Entities Measurement Results

The following percentage of test cases timed out when taking Context-Dependent A.C. into account:

Type \ # Scenarios	1x	2x	3x	4x	5x
Deassign Permission	-	-	28.2% (29)	31.1% (32)	45.6% (47)

Table 8.3: Timeouts Deassign Entities Test Cases and Context-Dependent A.C. and Increasing Number of Scenarios

Add/Remove Temporal Grant Rules We measured the time required to reverify the model after the addition of a temporal grant rule which grants a new demarcation (1), addition of a temporal grant rule which revokes a previously granted demarcation (2) and removal of a temporal grant rule (3). For (1), for each non-proxy role, we tried adding (up to 10) new temporal grant rule which permanently grants a new non-proxy demarcation to that role individually. For (2), for each rule which grants a non-proxy demarcation to a non-proxy role, we tried adding a rule which permanently revokes a granted demarcation from this role individually. For (3), we tried removing each temporal grant rule individually. For (1)-(3), we respectively had 209, 226 and 613 different test cases. However, since (1)-(3) can become quite costly in terms of time when taking Context-Dependent A.C. into account and when scaling up the model, we only performed 80 each on scaled-up versions of the model.

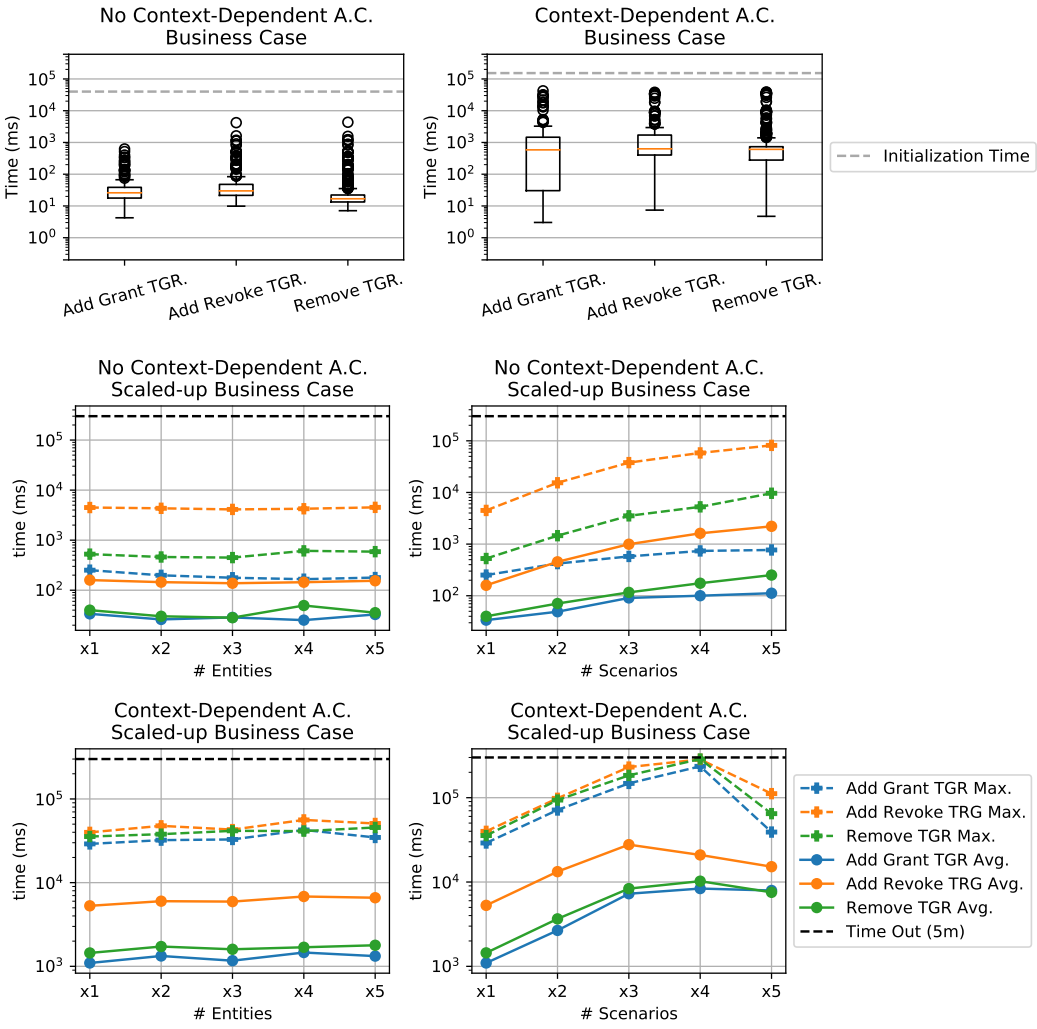


Figure 8.8: Add/Remove Temporal Grant Rules Measurement Results

The following percentage of test cases timed out when taking Context-Dependent

A.C. into account:

Type \ # Scenarios	1x	2x	3x	4x	5x
Add Grant TGR.	-	-	-	-	1.3% (1)
Add Revoke TGR.	-	-	-	7.5% (6)	11.3% (9)
Remove TGR.	-	-	-	-	2.5 % (2)

Table 8.4: Timeouts Add/Remove TGRs. with Context-Dependent A.C. and Increasing Number of Scenarios

Add/Remove Temporal Authentication Rules We measured the time required to reverify the model after the addition/removal of a temporal grant rule which sets a security zone's status to locked (1)/(2), unlocked (3)/(4) or protected (5)/(6). For (1), (3) and (5) for each temporal authentication rule which sets a certain status, we created two new rules with a higher priority which sets another status during the same temporal context. E.g., if the system has a rule which says the security zone Lobby should be Unlocked during Working Hours, we create two new rules which states this security zone should be locked and protected during the same temporal context with a higher priority. For (2), (4) and (6) we tried removing each temporal grant rule individually. For (1)-(6), we respectively had 448, 2, 160, 290, 292 and 158 different test cases. As (1)-(6) can become quite costly in terms of time when taking Context-Dependent A.C. into account and when scaling up the model, we only performed up to 80 each on scaled-up versions of the model. Since the temporal authentication rules are only taken into account when verifying context-dependent authorization constraints, we did not measure the impact of adding/removing temporal authentication rules when we do not take those kind of constraints into account.

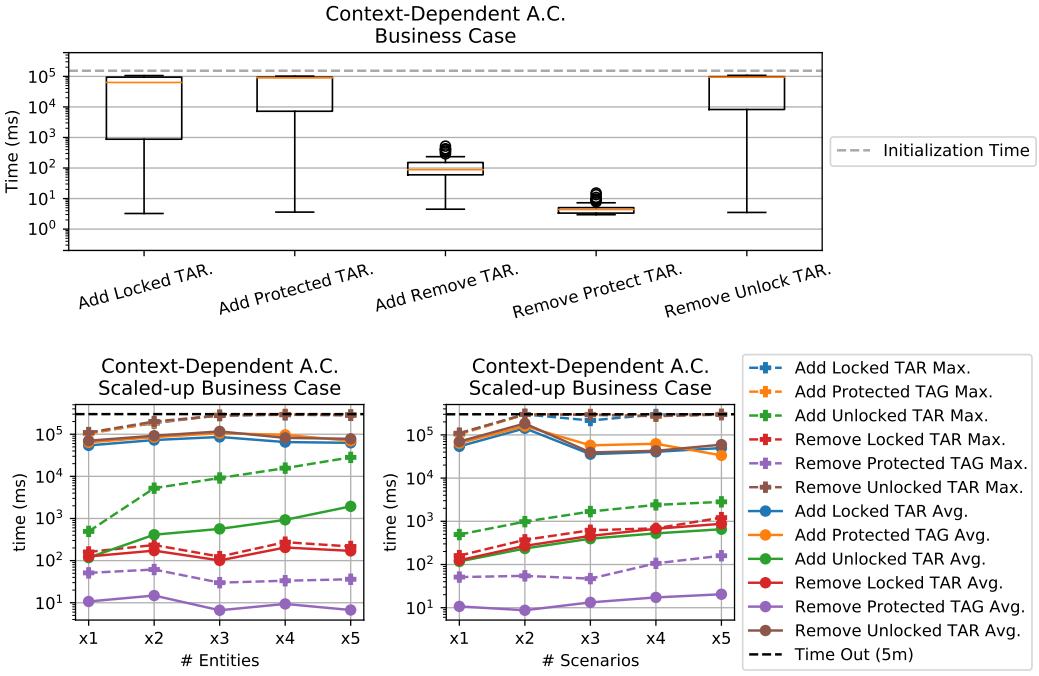


Figure 8.9: Add/Remove Temporal Authentication Rules Measurement Results

The following percentage of test cases timed out when taking Context-Dependent A.C. into account:

Type \ # Entities.	1x	2x	3x	4x	5x
Add Lock TAR.	-	-	7.5% (6)	25% (20)	28.8% (23)
Add Protect TAR.	-	-	10% (8)	32.5 % (26)	46.3 % (37)
Remove Unlock TAR.	-	-	7.5% (6)	41.3 % (33)	46.3 % (37)

Table 8.5: Timeouts Add/Remove TARs. with Context-Dependent A.C. and Increasing Number Of Model Entities

Type \ # Scenarios.	1x	2x	3x	4x	5x
Add Lock TAR.	-	8.8% (7)	52.5% (42)	55% (44)	55% (44)
Add Protect TAR.	-	21.3 % (17)	62.5 % (50)	65% (52)	72.5 % (58)
Remove Unlock TAR.	-	12.5% (10)	67.5% (54)	68.8% (55)	68.8% (55)

Table 8.6: Timeouts Add/Remove TARs. with Context-Dependent A.C. and Increasing Number Of Scenarios

Add/Remove Temporal Contexts We measured the time required to reverify the model after the creation of a new temporal context (1) and the removal of an existing temporal context (2). For (1), for each existing scenario, we tried adding a new temporal context which overlaps only with that scenario through one temporal context instance on the level of the day of the week/month. To measure (2), we tried removing each temporal context individually. For (1) and (2), we respectively had 40 and 38 different test cases.

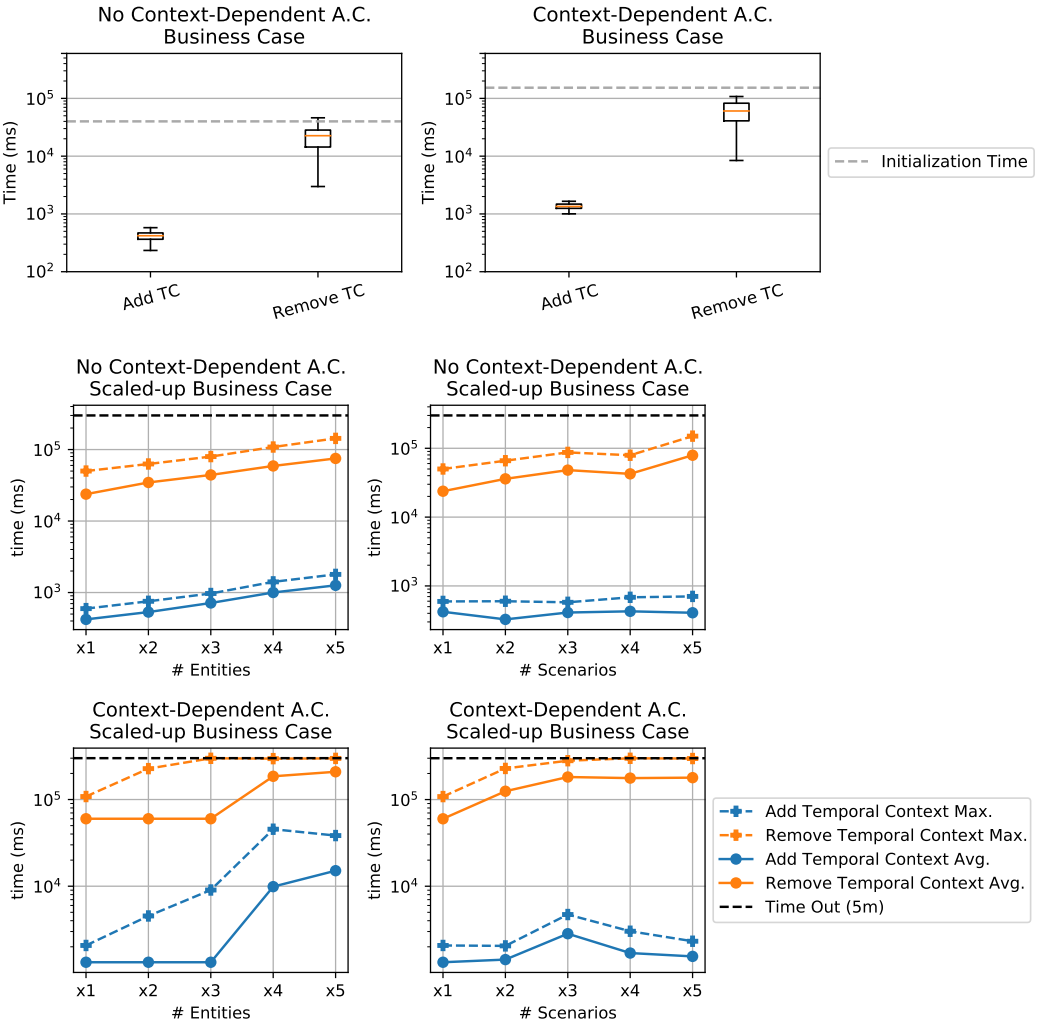


Figure 8.10: Add/Remove Temporal Contexts Measurement Results

The following percentage of test cases timed out when taking Context-Dependent A.C. into account:

Type \ # Entities	1x	2x	3x	4x	5x
Remove Temporal Context	-	-	18.4 % (7)	60.5% (23)	78.9 % (30)

Table 8.7: Timeouts Add/Remove TC. with Context-Dependent A.C. and Increasing Number of Model Entities

Type \ # Scenarios	1x	2x	3x	4x	5x
Remove Temporal Context	-	-	28.9 % (11)	31.6% (12)	55.3% (21)

Table 8.8: Timeouts Add/Remove TC. with Context-Dependent A.C. and Increasing Number of Scenarios

Add Authorization Constraints

We measured the time required to reverify the model after the addition of a policy-dependent authorization constraints which has at least one violation. We respectively generated up to 30 different test cases per policy-dependent authorization constraint subtype.

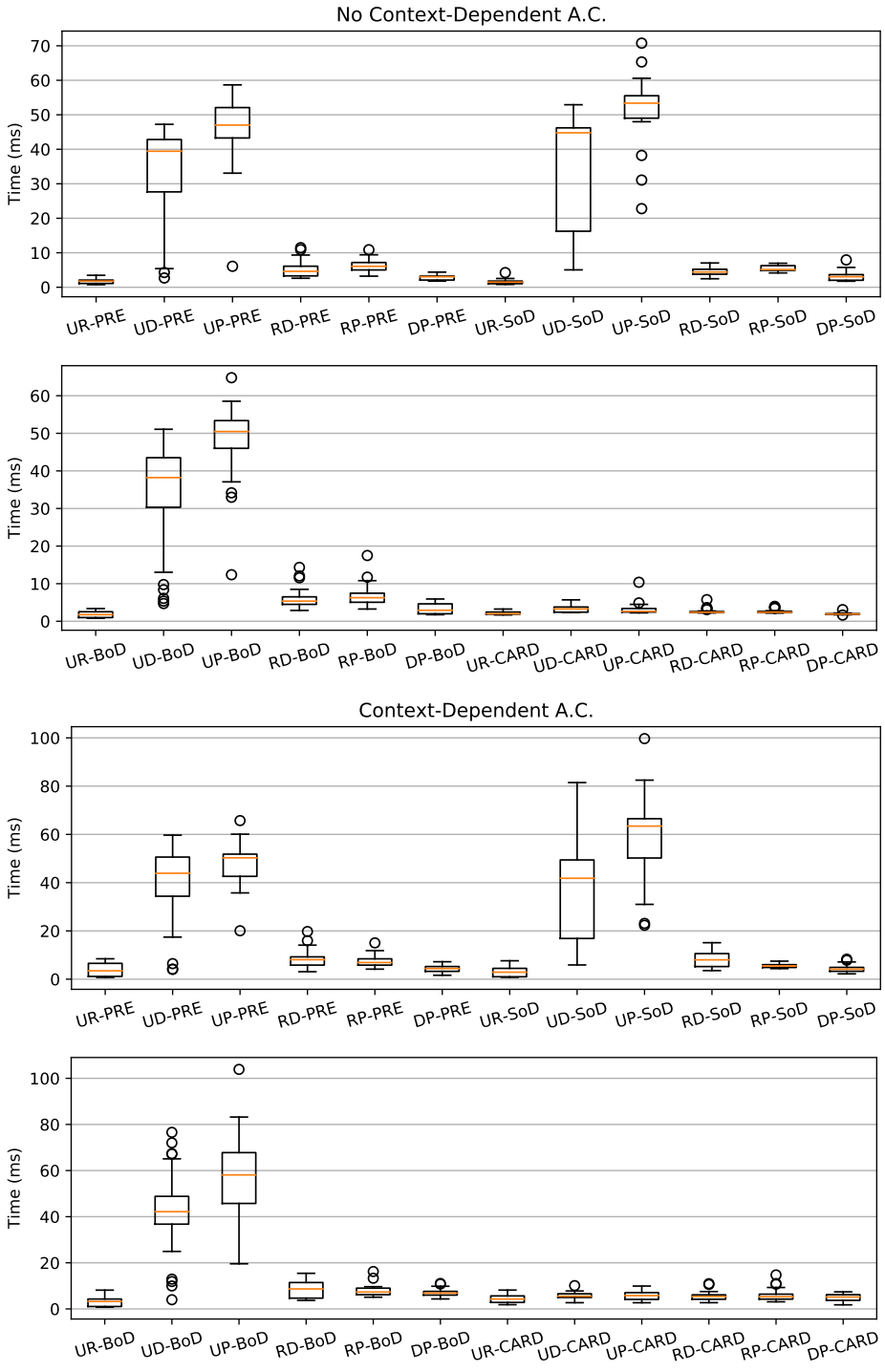
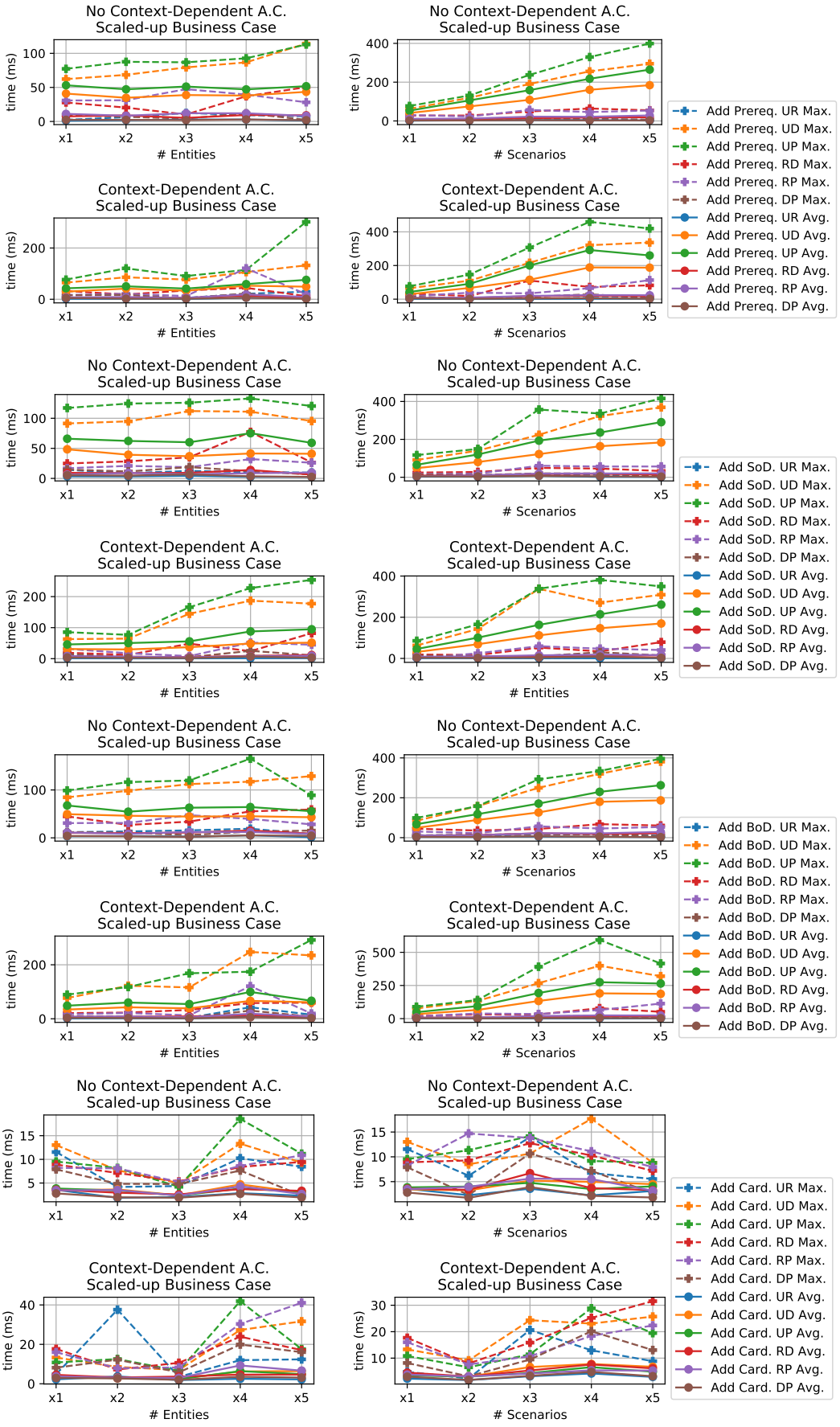


Figure 8.11: Add Policy-Dependent A.C. Measurement Results


No Context-Dependent A.C.
Scaled-up Business Case
time (ms)
Entities
Scenarios
Context-Dependent A.C.
Scaled-up Business Case
Add Prereq. UR Max.
Add Prereq. UD Max.
Add Prereq. UP Max.
Add Prereq. RD Max.
Add Prereq. RP Max.
Add Prereq. DP Max.
Add Prereq. UR Avg.
Add Prereq. UD Avg.
Add Prereq. UP Avg.
Add Prereq. RD Avg.
Add Prereq. RP Avg.
Add Prereq. DP Avg.
Add SoD. UR Max.
Add SoD. UD Max.
Add SoD. UP Max.
Add SoD. RD Max.
Add SoD. RP Max.
Add SoD. DP Max.
Add SoD. UR Avg.
Add SoD. UD Avg.
Add SoD. UP Avg.
Add SoD. RD Avg.
Add SoD. RP Avg.
Add SoD. DP Avg.
Add BoD. UR Max.
Add BoD. UD Max.
Add BoD. UP Max.
Add BoD. RD Max.
Add BoD. RP Max.
Add BoD. DP Max.
Add BoD. UR Avg.
Add BoD. UD Avg.
Add BoD. UP Avg.
Add BoD. RD Avg.
Add BoD. RP Avg.
Add BoD. DP Avg.
Add Card. UR Max.
Add Card. UD Max.
Add Card. UP Max.
Add Card. RD Max.
Add Card. RP Max.
Add Card. DP Max.
Add Card. UR Avg.
Add Card. UD Avg.
Add Card. UP Avg.
Add Card. RD Avg.
Add Card. RP Avg.
Add Card. DP Avg.

8.3 Analysis of results

From the results presented in the previous section, we can draw the following conclusions about our prototype:

Incremental verification provided a significant performance increase over batch verification. In general, incremental verification manages to drastically reduce the reverification time of the business case after a model change (under our assumptions).

Initialization time/memory usage generally seems to scale linearly and looks highly correlated. It looks as the initialization time and memory required to initialize the Rete engine scale linearly when the size of the model and the number of scenarios are scaled linearly (under duplication). The only exception seems when we increase the amount of model entities while not taking context-dependent authorization constraints into account. They also seem much more correlated than we initially anticipated. More research would be required to draw definitive conclusions.

Even though the Rete engine quite aggressively caches intermediary results, the memory footprint of the prototype is manageable on modern machines. This is the case even when the business case is scaled up to three-five times the amount of authorization/authentication model entities or the number of scenarios.

Verifying context-dependent authorization constraints currently has a high impact on performance. In general, it takes much longer to reverify the model after a model modification when we take context-dependent authorization constraints into account compared to when we do not. Furthermore, when we scale up the business case, we also start to observe many timeouts when we take context-dependent authorization constraints into account.

In the prototype, a recursive graph pattern is used to compute accessibility. This information is required to verify the context-dependent authorization constraints. The match set of this pattern is very expensive to initialize and update. When context-dependent authorization constraints are not taken into account, the match set of this pattern does not have to be initialized and updated. Together with the fact that some other patterns also do not have to be initialized, omitting context-dependent authorization constraints results in a drastic decrease in the initialization time, memory usage and the time required to reverify the model after many types of model changes.

The number of scenarios had a higher impact on performance than the amount of authorization/authentication model entities. Overall, the time required to reverify the model after a change increases faster when we increase the number of scenarios compared to when we increase the amount of authorization/authentication model entities.

Verifying new authorization constraints went almost instantly. All newly added authorization constraints added to the business case were verified in at most 110ms, often much faster. Even when scaling up the business case, new authorization constraints were verified in less than 600ms (and again, often much faster).

New user-demarcation and user-permission were the most expensive subtypes of prerequisite, SoD. and BoD. constraints to verify. Policy-dependent authorization constraints check if certain invariants are respected for one of the following relations: UR, USD, USP, RSD, RSP and DP. The greater the cardinality of these relations, the higher the number of computations a pattern checking invariants on one of these relations has to perform. User-demarcation and user-permission authorization constraints check invariants on the USD and USP relations. In the business case, the cardinality of USD and USP is much greater than that of UR, RSD, RSP and DP. Thus, authorization constraints on the user-demarcation and user-permission level were the most expensive to verify.

New cardinality constraints were inexpensive to verify. Even on the scaled-up business case, new cardinality constraints took on average less than 10ms and at most 45ms.

Not only the type but also the nature of the change has a high impact on the time required to reverify the model. For most model modifications, it holds that there is quite a high variation in the time which was required to reverify the model. Next to this, there are often quite a few outliers and quite a steep difference between the median time and the maximum which was required to reverify the model.

The current prototype is not fast enough for incremental continuous verification when taking context-dependent authorization constraints into account In contrast to batch verification, incremental verification is a blocking process. It is not possible to verify and analyze the impact of a new change before the impact of previous changes has been fully processed. When taking context-dependent authorization constraints into account, there are many model modifications which can take a signification amount of time. Thus, the achieved performance is not yet on the

level to support continuous verification and give security officers quick feedback after each possible model change. Furthermore, in the case the engine has to be reinitialized (e.g. because of a crash), a security officer must wait a significant amount of time before the engine is reinitialized.

The current prototype does not scale well enough for continuous verification when context-dependent authorization constraints are taken into account. When context-dependent constraints are taken into account, the time required to reverify the model after a change and the amount of memory used increases significantly when the model is scaled-up. Especially model modifications which have an effect on what many users can access (e.g. the removal of a much-used role/demarcation or setting a previously unlocked security zone to protected) take significantly longer with scaled-up models.

The currrent prototype shows incremental verification might be a promising avenue for continuous verification. Although the current prototype is not fast enough and does not scale well enough for continuous verification, it does show promise. When not taking context-dependent authorization constraints into account, the performance of increases significantly and the prototype scales relatively well. The model modifications a security officer is expected to perform most are the creation of users and the assignment/deassignment of roles to/from users and permission to/from demarcations. All measurements we performed of these type of model modifications at most 160ms on the non-scaled up business case. The measurements of the other model modifications which a security officer performs more rarely in most cases took less than 1 second on the non-scaled up business case. This is an important limit as 1 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay [71].

There were a few exceptions. We have measured that adding temporal grant rules which revoke grants, removing temporal grant rules and removing much-used roles/demarcations could take over 1 second. There is also the problem that reverifying the model after removing a temporal context often took almost as long as reinitializing the whole engine because of how we implemented the process to compute the set of scenarios. There are many areas of improvement, however the results of this initial prototype look promising. Thus, based on these results we conclude that incremental verification of physical access control merits future research.

Chapter 9

Final Thoughts

Conclusion

In this thesis, we explored a model, approach and prototype to incrementally verify various types of authorization constraints on realistic evolving physical access control systems.

Inspired by RBÄC and the ideas introduced by TRBAC, we presented and formalized Gregorian RBAC (GR-RBAC) as a possible model to capture physical access control authorization policies. We introduced the concept of demarcation granting/revoking inspired by role enabling/disabling and the concept of time ranges as an improvement upon periodic expressions. Together, these concepts allow for time-dependent accessibility.

We also introduced two simple formalisms which can capture physical access control topologies and authentication policies. We showed how GR-RBAC and these two formalisms can be linked in a single overarching formalism: the Site Access Control System (SACS) model.

Next, we presented how common authorization constraints can be defined on this model. We made a distinction between two types of authorization constraints: policy-independent and policy-dependent constraints. We also introduced the new concepts of policy smells and context-dependent authorization constraints.

We showed how the previously introduced formalisms can be translated into EMF/VIATRA. Next, we demonstrated that the various presented types of authorization constraints can be incrementally verified through incremental graph pattern match-

ing by building a prototype implementation with VIATRA.

To get an impression on how efficient and scalable the chosen approach is, we performed an exhaustive benchmark exploring many aspects of our prototype. We did this based on an actual physical access control policy of an anonymous company provided by Nedap. We publicly released the source code of our verification engine, an anonymized version of the provided real physical access control policy, a CLI version of the verification engine and all test cases on `https://github.com/HansvdLaan/GRRBAC-Verifier/`. We draw the conclusion that incremental verification often drastically speeds-up reverification. However, we also observe that the current prototype is not fast/scalable enough for continuous verification of actual physical access control systems. Nevertheless, we conclude that our approach shows promise.

Secondary Contributions

Our work contributed in improving the VIATRA project by spotting some bugs and limitations. We also had quite a few public discussions with people behind VIATRA on their forum regarding topics such as performance, the internal workings of the Rete engine and how to approach certain problems which increases the general knowledge base. We think that especially the discussions regarding performance would be useful for other VIATRA practitioners to read as there are some tips given which are not present in the documentation. Lastly, we think that the benchmark setup we have created with CRIU might be useful for the VIATRA project to replicate as it allows for very extensive and aggressive benchmarks.

We have also indirectly contributed to the Linux kernel. When we started with CRIU, we could not successfully restore JVMs. We reported this to the maintainers of CRIU. They discovered that this was because of a kernel bug which they subsequently fixed [1].

Limitations

We would like to point out the following limitations of our research:

Scenario computation is complex. The most important limitations of our research are related to the computation of the set of scenarios. The process to com-

[1] `https://lkml.org/lkml/2020/10/15/583` (accessed on 22-1-2021)

pute them has grown quite complicated and is interwoven with assumptions which hold for the Gregorian calendar but which may or may not hold for other calendars. We deem that two important directions of future research will be difficult to address given how we compute them currently. Namely, 1) how to support event-based contexts and 2) how to deal with scenario explosions. We will elaborate on both these areas of possible future work later on in this section.

We believe the process to compute the set of scenarios could be simplified by switching from an interval-based approach to an automata-based approach. If we could model when a temporal context occurs with (timed) automata, we could find all occurring scenarios by looking for all reachable states in the product automata of the temporal contexts.

Scenario computation implementation is inefficient. We have implemented the computation of scenarios in an inefficient way. Given how we compute scenarios, our prototype unnecessarily computes various relations and proves authorization constraints for spurious scenarios. Secondly, a lot of computations have to be redone when temporal contexts are removed since the modified scenario are registered as new scenarios.

No mechanism against scenario explosions. There is currently no mechanism in place which, akin to a state-space reduction, reduces the number of scenarios for which certain types of authorizations constraints have to be taken into account. As can be concluded from the benchmark, the absence of a scenario reduction mechanism is an import limit to the scalability of our current prototype. Such a reduction could provide a significant speed-up and reduce the impact of the number of scenarios on the performance of the analysis engine.

We believe there is ample opportunity for such reduction as we expect that from the perspective of a user and role, many system-wide scenarios could be considered equivalent. Unfortunately, finding which scenarios are equivalent from e.g. the perspective of a user is trickier than it might initially look. From an authorization perspective, it is trivial: two scenarios can be considered equivalent for a given user if the intersection of the sets of temporal contexts associated with these scenarios with the set of temporal contexts which are part of temporal grant rules which grant/revoke a demarcation to a role the user is assigned are equal. However, from an authentication perspective, it is not so trivial as which temporal authentication rules have an impact depends on what zones a user can reach, which (partly) depends on the temporal authentication rules.

Only security zones are taken into account as access-controlled objects. From discussions with security officers and based on our business case, we must conclude that having only security zones as objects on which access can be controlled is too simple to either model and/or support actual physical access control policies. Next to security zones, security officers want to define measures, obligations and perform checks on individual doors and door sides. Additionally, alarms, lockers and door radars are also important physical access control entities which are currently not taken into account.

Computing accessibility is too expensive. The context-dependent authorization constraints are currently verified by looking if there exists a scenario in which a user can not reach or leave a particular security zone during a given scenario. This is computed with the help of recursive graph patterns which computes this rather naively. Initializing and updating these patterns have proven to be quite expensive in practice. We believe another approach is required to compute reachability and/or verify the context-dependent authorization constraints. We believe a fruitful approach might be to enforce reachability by construction through demarcations. To illustrate, this might be done by enforcing for each demarcation that there is a path in each direction between all assigned/inherited security zones. Another approach would be to enforce reachability on the level of roles instead of on the level of users, similar to [63].

Memory trade-off between incremental and batch verification was not evaluated. Due to the way we implemented our prototype, VIATRA's non-incremental local search engine was unreasonably inefficient when verifying authorization constraints. Thus, we have used VIATRA's incremental Rete engine as a stand-in batch engine during our evaluation. However, this engine is incremental by design; the Rete networks the Rete engine builds and fills to compute the matches for each graph pattern also serve as the cache. Thus, we were unable to get a faithful impression about the additional amount of memory required (for the cache) when performing incremental verification instead of batch verification.

Upscaled business case not representative for larger access controlled systems. We would also like to point out that, although the scaled-up versions of our business case give us an impression about the scalability of our prototype, they are not representative of actual access control systems of larger companies or larger physical access control systems. Based on conversations with security experts from Nedap, it is suspected these will be structured differently. Namely, it is suspected

that they will be less granular. Thus, although there will be more users and permissions, it is expected that the number of roles and demarcations will be relatively lower. Also, the number of temporal contexts is suspected of growing only slowly when the size of the company and the access control increases. However, more research would be required to confirm this. Thus, we do not know if our solution would be scalable enough for bigger physical access control systems.

Future Work

Next to addressing the limitations, we consider the following research directions to be of interest for future work.

Increase Robustness Moving from limitations to opportunities, we believe a feature which could help facilitate policy evolution is a mechanism which can withhold (previously granted) permissions. An extension to RBÄC called polarized RBÄC was presented in the same paper RBÄC was introduced in a which offers this exact functionality [1]. We thus believe it could be worthwhile to investigate into combining polarized RBÄC and TRBAC.

To illustrate, take the use-case that, for a specific day, a zone should not be accessible by a group of roles (e.g. for an event). Currently, there is no direct way to state directly that individual permission should be revoked from a role during a specific context. Processing this single, and arguably simple, requirement might require many new temporal grant rules have to be defined for that specific day which revoke existing grants and define new grants. Specifically, a new temporal grant rule has to be created for each existing temporal grant rule, which grants that demarcation to a role which should temporarily be denied that zone, that overwrites it for that specific day. Furthermore, a copy of all demarcations containing this zone except the zone itself has to be made which has to be granted again to all roles which were denied the one specific zone during the moments they are denied the original demarcations. Processing this requirement would have been much simpler if it would be possible to withhold previously granted permissions.

Another feature which we argue could perhaps make the policy more robust are temporal constraints which grant/revoke demarcations to/from roles based upon which demarcations are also being granted/revoked to/from that role. As we showed in 4.3.2, this could improve policy conciseness and understandability. However, it does complicate the conflict detection and resolution process significantly. Furthermore, it

might be confusing for a security officer if something happens automatically when a demarcation is granted to/revoked from a role. Nonetheless, if one wants to add this kind of temporal constraints, we believe a good jumping-off point to start with this feature would be to see if the graph-based algorithm to detect temporal constraint contradictions in TRBAC can be incrementalized.

Role granting/revoking Three common use-cases in physical access control are that users are granted a new role for one or a few days (e.g. for events), until a certain date (e.g. the end of their contract) or occupy a role intermittently (e.g. because they work part-time or in shifts). All three use-cases were present in the business case, although only the first two were reflected in the access control policy.

Unfortunatly, these use-cases are not directly supported by our proposed authorization model. GR-RBAC does not support granting and revoking of roles to users based on the temporal context. All three use-cases could be encoded in the model by creating copies of roles and modifying the set of temporal grant rules attached to these rules. However, supporting the use-cases in this way is arguably against the spirit of the formalism we inspire ourselves upon, RBÄC, as roles should represent abstractions over the set of users which are not tied to the implementation of the access control policy.

Event-based Contexts The presented GR-RBAC formalism does not support event-based contexts. In physical access control, these kind of contexts are often implicitly or explicitly part of the authorization and authentication model. With regards to authorization, physical access control policies often have a group of roles corresponding to various emergency service personnel roles which gain permissions during a calamity. With regards to authentication, a common security practice is that the front door is protected with access control measures unless when the receptionist is at her workstation. In that case, visitors can often enter the lobby freely. Both cases were part of the provided anonymized access control policy, although the first use-case was performed as a process outside of the access control system itself. To effectively capture these parts of physical access control policies, event-based contexts need to be supported.

The idea of contextual/reactive policies has been explored before. In [72], the authors present ERBAC, a formalism which extends TRBAC with events and environmental conditions. Another formalism, GRBAC [73], introduced the concept of an environmental role which could automatically be activated/deactivated based on environmental conditions. In [74], the CAAC formalism was presented, which sup-

ports context-aware user-role and role-permission assignments. For an extensive overview of what has been done regarding reactive policies, we refer the reader to [74].

Hybrid Checking VIATRA can combine both incremental and non-incremental techniques when evaluating a set of graph patterns: for each individual graph pattern, it can be indicated if the match set should be computed using the Rete engine or the local search engine. Prior work has been done combining both techniques [75]. We did not use this functionality during our research. In future work, it could be interesting to explore this feature and evaluate the loss in performance compared to the memory saved by using the local search engine for certain patterns instead of the Rete engine.

Policy Repair When authorization constraint violations and policy smells have been detected, it would be useful for a security officer to know how they could be resolved. This corresponds to the problem of model repair: what is a set of model transformations that "repair" the model such that it conforms to all constraints? VIATRA has been used before to generate so-called quick fixes for domain-specific modelling languages [76]. For access control, this question has also been explored before in the domain of XML write-access control policies [77] and for XACML policies [78]. We believe it could be interesting to investigate if these techniques would also be appropriate for the domain of physical access control or if this domain introduces new challenges.

Path-Informed Verification We have observed in our business case that permissions to access security zones are given because of two reasons: either the role has direct business in the security zone, or they are given to facilitate paths and short-cuts. At the moment, this intent is not captured in the model. We believe that it could be worthwhile to investigate what new possibilities open up if security officers can specify this intent. To start, we believe it could allow for path-based verification such as verifying that all paths conform to the principle of least privilege. We can draw inspiration from [63]. Here, the authors proposed to support security officers by suggesting paths conforming to various criteria such as least privilege or shortest distance.

Bibliography

[1] W. Kuijper and V. Ermolaev, “Sorting out Role based access control,” in *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, 2014.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-based access control models,” 1996.

[3] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, A. Russo, and C. Williams, “Methods and Tools for Policy Analysis,” *ACM Computing Surveys*, vol. 51, no. 6, pp. 1–35, feb 2019. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3303862.3295749

[4] M. I. Gofman, R. Luo, J. He, Y. Zhang, and P. Yang, “Incremental Information Flow Analysis of Role Based Access Control,” *Security and Management*, pp. 397–403, 2009.

[5] M. I. Gofman and P. Yang, “Efficient Policy Analysis for Evolving Administrative Role Based Access Control,” *Int. J. Software and Informatics*, vol. 8, no. 1, pp. 95–131, 2014.

[6] S. Ranise and A. Truong, “Incremental Analysis of Evolving Administrative Role Based Access Control Policies,” ser. Lecture Notes in Computer Science, V. Atluri and G. Pernul, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8566, pp. 260–275. [Online]. Available: http://link.springer.com/10.1007/978-3-662-43936-4http://link.springer.com/10.1007/978-3-662-43936-4{_}17

[7] M. Vierhauser, D. Dhungana, W. Heider, R. Rabiser, and A. Egyed, “Tool Support for Incremental Consistency Checking on Variability Models,” *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, 2010.

[8] H. Lu, T. Yue, S. Ali, and L. Zhang, “Model-based incremental conformance checking to enable interactive product configuration,” *Information and Software Technology*, 2016.

[9] A. Reder and A. Egyed, “Incremental consistency checking for complex design rules and larger model changes,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[10] G. to graph patterns;bor Bergmann, “Translating OCL to graph patterns,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014.

[11] Y. Papakonstantinou and V. Vianu, “Incremental validation of XML documents,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2003.

[12] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas, “Efficient incremental validation of XML documents,” in *Proceedings - International Conference on Data Engineering*, 2004.

[13] T. Szabó, S. Erdweg, and M. Voelter, “IncA: A DSL for the definition of incremental program analyses,” in *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[14] L. Sion, D. Van Landuyt, K. Yskout, and W. Joosen, “SPARTA: Security & Privacy Architecture Through Risk-Driven Threat Assessment,” in *Proceedings - 2018 IEEE 15th International Conference on Software Architecture Companion, ICSA-C 2018*, 2018.

[15] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, “Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework,” *Software and Systems Modeling*, 2016.

[16] G. Bergmann, C. Debreceni, I. Ráth, and D. Varró, “Query-based access control for secure collaborative modeling using bidirectional transformations*,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems - MODELS ’16*. New York, New York, USA: ACM Press, 2016, pp. 351–361. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976767.2976793

[17] C. Debreceni, G. Bergmann, I. Ráth, and D. Varró, “Enforcing fine-grained access control for secure collaborative modelling using bidirectional transformations,” *Software and Systems Modeling*, 2019.

[18] A. Karp, H. Haury, and M. Davis, “From ABAC to ZBAC: The evolution of access control models,” in *5th European Conference on Information Management and Evaluation, ECIME 2011*, 2011.

[19] V. C. V. Hu, D. F. Ferraiolo, and D. R. Kuhn, “Assessment of access control systems,” *Nistir 7316*, 2006.

[20] E. Conrad, S. Misenar, and J. Feldman, *Eleventh Hour CISSP®: Study Guide*, 3rd ed. Syngress, 2016.

[21] Committee on National Security Systems, “Security and Privacy Controls for Federal Information Systems and Organizations,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., apr 2013. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf

[22] S. Osborn, R. Sandhu, and Q. Munawer, “Configuring role-based access control to enforce mandatory and discretionary access control policies,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 2, pp. 85–106, may 2000. [Online]. Available: http://dl.acm.org/doi/10.1145/354876.354878

[23] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca, “GEO-RBAC: A spatially aware RBAC,” in *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, 2005.

[24] E. Bertino, P. A. Bonatti, and E. Ferrari, “TRBAC: A Temporal Role-Based Access Control Model,” *ACM Transactions on Information and System Security*, 2001.

[25] J. B. Joshi, E. Bertino, U. Latif, and A. Ghafoor, “A generalized temporal role-based access control model,” *IEEE Transactions on Knowledge and Data Engineering*, 2005.

[26] I. Ray and M. Toahchoodee, “A spatio-temporal role-based access control model,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2007.

[27] S. Arjmand, A. Ghafoor, and E. Bertino, “A Framework for Specification and Verification of Generalized Spatio-Temporal Role based Access Control Model,” *CERIAS Tech Report*, 2007.

[28] S. Oh and S. Park, “Task-role-based access control model,” *Information Systems*, 2003.

[29] L. Wang, D. Wijesekera, and S. Jajodia, “A logic-based framework for attribute based access control,” in *FMSE’04: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, 2004.

[30] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, “Attribute-based access control,” *Computer*, 2015.

[31] D. R. Kuhn, E. J. Coyne, and T. R. Weil, “Adding attributes to role-based access control,” *Computer*, 2010.

[32] A. Ben Fadhel, D. Bianculli, and L. Briand, “A comprehensive modeling framework for role-based access control policies,” *Journal of Systems and Software*, 2015.

[33] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo, “EXAM: A comprehensive environment for the analysis of access control policies,” *International Journal of Information Security*, 2010.

[34] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, “Mining roles with multiple objectives,” *ACM Transactions on Information and System Security*, 2010.

[35] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, “Graph pattern matching: From intractable to polynomial time,” *Proceedings of the VLDB Endowment*, 2010.

[36] J. R. Falleri, X. Blanc, R. Bendraou, M. A. A. Da Silva, and C. Teyton, “Incremental inconsistency detection with low memory overhead,” in *Software - Practice and Experience*, 2014.

[37] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, “Incremental evaluation of model queries over EMF models,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

[38] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, “EMF-IncQuery: An integrated development environment for live model queries,” *Science of Computer Programming*, 2015.

[39] Z. Ujhelyi, G. Szoke, Á. Horváth, N. I. Csiszár, L. Vidács, D. Varró, and R. Ferenc, “Performance comparison of query-based techniques for anti-pattern detection,” in *Information and Software Technology*, 2015.

[40] Z. Ujhelyi, “Program analysis techniques for model queries and transformations,” Ph.D. dissertation, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2016.

[41] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, “A graph query language for EMF models,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011.

[42] M. Búr, Z. Ujhelyi, Á. Horváth, and D. Varró, “Local search-based pattern matching features in EMF-INCQUERY,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[43] G. Bergmann, “Incremental model queries in model-driven design,” Ph.D. dissertation, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2013.

[44] A. Calvi, S. Ranise, and L. Viganò, “Automated validation of security-sensitive web services specified in BPEL and RBAC,” in *Proceedings - 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010*, 2010.

[45] T. Kosiyatrakul, S. Older, and S. K. Chin, “A modal logic for role-based access control,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2005.

[46] F. Cuppens, N. Cuppens-Boulahia, M. B. Ghorbel-Talbi, S. Morucci, and N. Essaouni, “Smatch: Formal dynamic session management model for RBAC,” *Journal of Information Security and Applications*, 2013.

[47] L. Kahloul, K. Djouani, W. Tfaili, A. Chaoui, and Y. Amirat, “Modeling and verification of RBAC security policies using colored petri nets and CPN-tool,” in *Communications in Computer and Information Science*, 2010.

[48] F. Hansen and V. Oleshchuk, “Conformance checking of RBAC policy and its implementation,” in *Lecture Notes in Computer Science*, 2005.

[49] H. Ben Attia, L. Kahloul, S. Benhazrallah, and S. Bourekkache, “Using Hierarchical Timed Coloured Petri Nets in the formal study of TRBAC security policies,” *International Journal of Information Security*, 2019.

[50] S. Jha, S. Sural, J. Vaidya, and V. Atluri, “Security analysis of temporal RBAC under an administrative model,” *Computers and Security*, 2014.

[51] S. Mondal and S. Sural, “Security analysis of temporal-RBAC using timed automata,” in *Proceedings - The 4th International Symposium on Information Assurance and Security, IAS 2008*, 2008.

[52] B. J. Berger, C. Maeder, R. Wete Nguempnang, K. Sohr, and C. Rubio-Medrano, “Towards Effective Verification of Multi-Model Access Control

Properties," in *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies - SACMAT '19*. New York, New York, USA: ACM Press, 2019, pp. 149–160. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3322431.3325105

[53] G. J. Ahn and M. E. Shin, "Role-based authorization constraints specification using Object Constraint Language," in *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, 2001.

[54] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-based modeling language for model-driven security," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2002.

[55] H. Wang, Y. Zhang, J. Cao, and J. Yang, "Specifying Role-Based Access Constraints with Object Constraint Language," 2004, pp. 687–696. [Online]. Available: http://link.springer.com/10.1007/978-3-540-24655-8{_}75

[56] I. Ray, N. Li, R. France, and D. K. Kim, "Using UML to visualize role-based access control constraints," in *Proceedings of ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, 2004.

[57] K. Sohr, T. Mustafa, X. Bao, and G. J. Ahn, "Enforcing role-based access control policies in Web Services with UML and OCL," in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2008.

[58] Ç. Cirit and F. Buzluca, "A UML profile for role-based access control," in *Proceedings of the 2nd international conference on Security of information and networks - SIN '09*. New York, New York, USA: ACM Press, 2009, p. 83. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1626195.1626217

[59] K. Sohr, M. Kuhlmann, M. Gogolla, H. Hu, and G.-J. Ahn, "Comprehensive two-level analysis of role-based delegation and revocation policies with UML and OCL," *Information and Software Technology*, vol. 54, no. 12, pp. 1396–1417, dec 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584912001115

[60] M. KUHLMANN, K. SOHR, and M. GOGOLLA, "Employing UML and OCL for designing and analysing role-based access control," *Mathematical Structures in Computer Science*, vol. 23, no. 4, pp. 796–833, aug 2013. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0960129512000266/type/journal{_}article

[61] A. Baumgrass and M. Strembeck, “An approach to bridge the gap between role mining and role engineering via migration guides,” in *Proceedings - 2012 7th International Conference on Availability, Reliability and Security, ARES 2012*, 2012.

[62] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo, “A tool for domain-independent model mutation,” *Science of Computer Programming*, vol. 163, pp. 85–92, oct 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167642318300261

[63] N. Skandhakumar, F. Salim, J. Reid, and E. Dawson, “Physical access control administration using building information models,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[64] N. Skandhakumar, F. Salim, J. Reid, R. Drogemuller, and E. Dawson, “Graph theory based representation of building information models for access control applications,” *Automation in Construction*, 2016.

[65] A. Ben Fadhel, D. Bianculli, L. Briand, and B. Hourte, “A model-driven approach to representing and checking RBAC contextual policies,” in *CODASPY 2016 - Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, 2016.

[66] M. Niezette and J. Stevenne, “An efficient symbolic representation of periodic time,” in *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 1992, pp. 161–168.

[67] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, “An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning,” *ACM Transactions on Database Systems*, 1998.

[68] S. Jajodia, V. S. Subrahmanian, P. Samarati, and E. Bertino, “A Unified Framework for Enforcing Multiple Access Control Policies,” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 1997.

[69] Y. Sun, Q. Wang, N. Li, E. Bertino, and M. Atallah, “On the complexity of authorization in RBAC under qualification and security constraints,” *IEEE Transactions on Dependable and Secure Computing*, 2011.

[70] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, “The Train Benchmark: cross-technology performance evaluation of continuous model queries,” *Software and Systems Modeling*, 2018.

[71] J. NIELSEN, “Usability Heuristics,” in *Usability Engineering*, 1993.

[72] P. Bonatti, C. Galdi, and D. Torres, “Event-driven RBAC,” *Journal of Computer Security*, 2015.

[73] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd, “Securing context-aware applications using environment roles,” in *Proceedings of Sixth ACM Symposium on Access Control Models and Technologies (SACMAT 2001)*, 2001.

[74] A. S. M. Kayes, J. Han, W. Rahayu, T. Dillon, M. S. Islam, and A. Colman, “A Policy Model and Framework for Context-Aware Access Control to Information Resources†,” *The Computer Journal*, vol. 62, no. 5, pp. 670–705, may 2019. [Online]. Available: https://academic.oup.com/comjnl/article/62/5/670/5055357

[75] Á. Horváth, G. Bergmann, I. Ráth, and D. Varró, “Experimental assessment of combining pattern matching strategies with VIATRA2,” *International Journal on Software Tools for Technology Transfer*, 2010.

[76] Á. Hegedus, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró, “Quick fix generation for DSMLs,” in *Proceedings - 2011 IEEE Symposium on Visual Languages and Human Centric Computing, VL/HCC 2011*, 2011.

[77] L. Bravo, J. Cheney, I. Fundulaki, and R. Segovia, “Consistency and repair for XML write-access control policies,” *VLDB Journal*, 2012.

[78] D. Xu and S. Peng, “Towards automatic repair of access control policies,” in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, dec 2016, pp. 485–492. [Online]. Available: http://ieeexplore.ieee.org/document/7907003/