2022-02

Higher-Order (Temporal) Relationship-Based Access Control

Arora, Chahal

UNIVERSITY OF CALGARY

Higher-Order (Temporal) Relationship-Based Access Control

by

Chahal Arora

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

FEBRUARY, 2022

Abstract

With the advent of technologies such as the Internet of Things, new type of relationships have emerged between users and devices. These relationships are transient, which means they can be activated and terminated over time. Existing Relationship-Based Access Control (ReBAC) models are not designed for handling such relationships efficiently. In this work, we present a ReBAC model that can incorporate such transient relationships, thus allowing the creation of access control policies that can use the transient nature of relationships to grant authorization. We call this model Higher-Order (Temporal) Relationship-Based Access Control (HO(T)-ReBAC) model. This thesis formalized the HO(T)-ReBAC model and defined a formal policy language for access control policies in HO(T)-ReBAC. We then discussed case studies based on real-world scenarios where HO(T)-ReBAC can be deployed for authorization decisions. After that, we designed and presented an efficient model implementation that can be used for large-scale projects in the real world. We empirically evaluated our implementation of HO(T)-ReBAC using a real-world social graph and the use case we discussed. Our evaluation found our implementation to be efficient for real-world large-scale projects.

Acknowledgements

My journey of writing this thesis has been a great learning experience, and hence, I would like to extend my sincere gratitude to the few most prominent people in my life who stood by me in this pathway. First and foremost, I would like to thank my supervisor Dr. Philip W.L. Fong, for his guidance, mentorship, enthusiasm, and encouragement. The person who empowered me with his immense knowledge and taught me the importance of deep work. I would also like to thank the committee members Dr. Joel Reardon and Dr. Simon Li for spending their valuable time to read my thesis.

My profound gratitude to my parents, Rakesh Arora and Sunita Arora, for being my pillar of strength and giving me the vision to achieve what I always aspire for. I would also like to thank my wife, Akshita Rishi, for being super supportive, my elder brother, Sumit Arora, for always believing in me and encouraging me to bring out my best version in everything I do, and my sister-in-law, Sakshi Chopra, for always standing by my side. I would also express my thanks to my friends, my teachers, and all my loved ones for their best wishes, teachings, and blessings.

This thesis is a tribute to my mother, Sunita Arora, who not only had always been a great mother but my first teacher, my friend, my guiding force, and my support system. She always ensured that I had access to everything I needed to achieve my goals. Her commitment to my education and my well-being has been the most significant factor in me achieving all that I have in my life today. I would like to recount my limitless reverence for her "never give up" spirit and the way she fearlessly fought a long battle of cancer for 18 years, that too with a big bright smile on her face. She wanted to see me holding this thesis, and hence, I dedicate all my hard work to my mother. I miss you Mom, will always strive to make you proud.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

1.1 Overview

Access Control ensures that the actions that a user can perform in a multi-user system are regulated in accordance to a policy. Role-Based Access Control (RBAC) is one of the widely used access control models. RBAC has been around for decades, and so over the years, many variants of RBAC have come up for different use cases. For example, some variants may include the history of the system [13, 22, 21], along with the role of a user to make the final authorization decision. Relationship-Based Access Control (ReBAC) [14, 17, 30, 26] is another general-purpose access control model that has been relatively new. In ReBAC, the authorization decisions are based on the relationships between entities such as users, resources or groups of the system. In this thesis, we introduce Higher-Order (Temporal) Relationship-Based Access Control (HO(T)-ReBAC), an access control model that introduces a temporal dimension into ReBAC.

As the world becomes more interconnected new use cases on how users interact with devices and with each other are now emerging. The introduction of the Internet of Things or IoT, a network of connected devices sharing data frequently, has also brought up new information-sharing scenarios. An exciting concept that has come up in IoT interactions is *transient relationships* between the entities. A transient relationships is a relationship between the entities that can be activated and terminated over time.

Some examples for transient relations are (i) *colocation*, (ii) *scheduling*, and (iii) *subscription*. *colocation* describe presence of two IoT devices in the same physical location. Being present at the exact physical location can allow these devices to share information. Once these devices are no longer present at the same location, they might not need to share the information, so their interaction can be seen as terminated. In *scheduling*, a user can access a device during their working

hours. Therefore the interactions between the users and the device occur in a schedule. Similarly, *subscription* lets a user access some device for the time periods when the user has subscribed to a certain service. There is a temporal dimension between the user and the device in all three scenarios. This temporal aspect can affect the decision on whether a user should be able to access a device.

ReBAC can capture the interactions between the entities through relationships. Now suppose we would like to make the authorization decision based on the active as well as terminated relationships between the entities. In ReBAC, authorization takes into account only the currently active relationships; thus, how entities are historically related to one another is not considered in the authorization decision. However, in the model proposed in this thesis, we can capture these terminated as well as active relationships between the entities to make authorization decisions. CSC proposed by Fong *et al.* [16] also introduced the temporal dimension in ReBAC. The proposed policy language is highly expressive, but the inefficiency of its authorization algorithm restricted its usefulness for large-scale projects. In HO(T)-ReBAC, we focused on the real-world application use of the model and designed it such that the implementation is efficient for large-scale projects. Not only is HO(T)-ReBAC designed to be efficient, but it can also handle complexities. In HO(T)-ReBAC, a user can also access a device through a complex composition of relationships. The way we make authorization decisions in HO(T)-ReBAC differs from the way authorization decisions are made in ReBAC. In HO(T)-ReBAC, authorization decision are based on the temporal relations between the relationships in the system whereas in ReBAC authorization decisions are based on the relationships the resource owner has with the resource accessor.

The "higher-order" in HO(T)-ReBAC comes from the idea that authorization decisions are based on how the relationships are related to each other in this model. The "temporal" in HO(T)-ReBAC is there because we check the temporal relations between the relationships. We check for temporal relations between relationships because in HO(T)-ReBAC we are working with transient relationships. Since it is a relationship-based access control model hence the "ReBAC" part of

the name. Therefore HO(T)-ReBAC is a relationship-based access control model in which the authorization decisions are based on the temporal relations between the transient relationships.

1.2 Contribution

The first contribution for this work is that we developed and formalized an access control model that utilizes transient relationships for authorization decisions. We expanded the existing ReBAC model to incorporate history of how entities are related to one another. An example policy for a ReBAC system like Facebook can be that "only friends of a user can the users' post". This policy gives access to the users who currently have a "friend" relationship with the user. However, suppose we need policies such that we want to provide access to the user who currently has a "friend" relationship or those who had a "friend" relationship with the user at some past time point. One more example where this need becomes quite apparent is the Whatsapp chat group. For example, a user who is a chat group member can access that group's information. However, when a user leaves the group, it may still want to keep access to the information that was part of the group during which the user was also a member of the chat group. The policies of ReBAC cannot address these complex but relatable scenarios. We will be discussing more complex use case scenarios that would require the history tracking of a ReBAC system. The Higher-Order(Temporal) Relationship-Based Access Control (HO(T)-ReBAC) incorporates history with a ReBAC model and can handle policies for such complex scenarios.

The second contribution for this work is that we demonstrated the utility of the HO(T)-ReBAC model. We investigated use cases in the real world where the model can be implemented for authorization decisions. We did that by exploring the transient relationships that can be found in the real world. We then investigated a use case based in a hospital environment containing IoT devices that can utilize HO(T)-ReBAC for authorization decisions. We showed the different scenarios in access control for such a hospital and leveraged HO(T)-ReBAC policies for such scenarios. After exploring a concrete use case for HO(T)-ReBAC, we investigated the use case for

HO(T)-ReBAC when the requestor and resource are part of a group. We showed that we could do this by emulating group operations in g-SIS [23] using access control policies in HO(T)-ReBAC.

The third contribution for this work is that we designed and presented an efficient implementation of HO(T)-ReBAC and thus making it feasible to deploy HO(T)-ReBAC in the real world for authorization decisions. We first showed that the problem of finding periods for the complex composition of relationships as a Constraint Satisfaction Problem (CSP). We refer to these periods for the complex composition of relationships as *official periods*. We will provide more details about CSP in the following chapters. By reducing the problem of finding official periods as a CSP, we could use existing algorithms to solve CSP. We further improved the efficiency of the existing CSP problems for finding official periods using constraints in our problem. We provided ways to cache these found official periods, which further improved the efficiency of the implementation. We also empirically evaluated our implementation for its efficiency. There are two sets of experiments that we conducted to evaluate the efficiency of our implementation. The first set of experiments evaluated the efficiency of the implementation in terms of finding the official periods. The second set of experiments evaluated the efficiency of the implementation in terms of getting the authorization decision using the found official periods.

1.3 Related Work

Relationship-Based Access Control is an access control model where access authorization decisions are based on whether the resource and requestor have a specific relation. The term Relationship-Based Access Control was independently coined by Gates [18] and Carminati and Ferrari [14]. ReBAC was formalized by Fong [17], who also showed the limitations of previous access control models such as Role-Based Access Control (RBAC) [33, 32] to express the policies that mimic the nature of these complex social computing. Rizvi *et al.* [26] combined Relationship-Based Access Control and Attribute-based Access control (ABAC) [20] to introduce AReBAC. AReBAC proposed a new query language called Nano-Cypher [26] based on Neo4j's Cypher language [5].

Nano-Cypher fused database queries with access control policies making the authorization efficient. In both ReBAC and AReBAC, there is no temporal dimension there no way to track the history of relationships, and hence authorization decisions are made only using the current state of relationships. In HO(T)-ReBAC, we introduced the temporal dimension to relationships and, therefore, can take the history of relationships into account to make authorization decisions giving HO(T)-ReBAC a flavour of History-Based Access Control (HBAC) [15, 34] with Relationship-Based Access Control (ReBAC) in an IoT environment.

HO(T)-ReBAC is not the first model to introduce a temporal dimension into ReBAC. CSC proposed by Fong *et al.* [16] also introduced a temporal dimension into ReBAC. Their proposed policy language is highly expressive, but the inefficiency of its authorization algorithm restricted its usefulness for large-scale projects. In HO(T)-ReBAC, we introduced the temporal dimension as well as presented an implementation to make authorization decisions that are efficient enough to be considered to have applications in the real world.

As we have described there has been little work that adds a temporal dimension to a ReBAC system before HO(T)-ReBAC. But there have been extensive work for introducing temporal dimension to RBAC has been done. Some of the example that are related to our are Temporal-RBAC (TRBAC) and Generalized Temporal-RBAC (GTRBAC). TRBAC proposed by Bertino *et al.* [13] added a temporal constraint to the Role-Based Access Control model (RBAC). In TRBAC, roles can be made available and unavailable through role enabling and disabling periodically and triggers. In HO(T)-ReBAC, instead of available and unavailable, we have relationships that becomes active and inactive. The authorization in TRBAC decision is always made based on the system's current state, unlike HO(T)-ReBAC, in which the system uses historical relationships between different participating entities (for example, subject, object, group) in the system to grant authorization permission. Joshi *et al.* [21] analyzed the effect of GTRBAC [22] temporal constraints on role hierarchy and introduced different temporal hierarchies. They showed the dynamism of hierarchies in temporal constraints. On the other hand in HO(T)-ReBAC, hierarchies are handle

using the complex composition of relationships.

Krishnan *et al.* [23] proposed the formal specification in Group-Centric Secure Information Sharing Model (g-SIS) for sharing resources in a group setting using linear temporal logic. g-SIS focuses on sharing objects that are entered or created inside a group by different users. g-SIS applies the temporal dimension but does not consider the relationships the users and objects share with the group. So in g-SIS, there is only one way for a user and object to be related to a group (i.e., by being part of the group), whereas since HO(T)-ReBAC is relationship based, a user or object can have different kinds of relations with a group. In HO(T)-ReBAC, periods and the relationships a user and object have with the group are considered for authorization decisions.

Pereira and Fong [24] introduced the SEPD model for resource sharing in an IoT environment. SEPD provides a means for strangers to gain trust without needing a global identity framework. In HO(T)-ReBAC, we are concerned about the access in entities such as users and resources related to each other. In SEPD, trust between entities is created using the history of presence in a location, whereas in HO(T)-ReBAC, the trust between entities comes from their relationships with each other. Authorizations in SEPD depend on the history of the presence of entities in a location, whereas in HO(T)-ReBAC, authorizations depend on the history of relationships and how the relationships are related to each other.

In this section, we have seen current work related to this thesis. In the next part, we will detail the organization of this thesis.

1.4 Organization

This thesis has been organized as follows:

1. Chapter 2 provides the background needed to understand this thesis.

2. Chapter 3 introduces an access control model, HO(T)-ReBAC, in which access is conditioned on the transient relationships. We first formalized the HO(T)-ReBAC model and then proposed a policy language for HO(T)-ReBAC policies.

3. Chapter 4 examines the usefulness of HO(T)-ReBAC in the real world. We first explored the type of transient relations that can be found in the real world. We then examined a concrete example of a smart hospital setting where HO(T)-ReBAC can be deployed for authorization decisions. We further investigated how HO(T)-ReBAC can be useful in a group setting.

4. Chapter 5 describes an efficient implementation for HO(T)-ReBAC. In this chapter, we designed algorithms to efficiently find the periods of materialization for the complex composition of relationships. We further presented ways to cache the found periods of materialization in a database. After this, we describe a translation from HO(T)-ReBAC policies to database query to find an authorization decision in HO(T)-ReBAC.

5. Chapter 6 empirically evaluates the performance of HO(T)-ReBAC. We did this by dividing the evaluation into two parts. First, we evaluated our algorithms for finding periods of materialization, and after that, we evaluated the performance of the database queries we generated from the HO(T)-ReBAC policy to find authorization decisions.

6. Chapter 7 provides the conclusion and future work for this thesis.

This chapter gives a brief introduction about a new access control model namely Higher-Order (Temporal) Relationship-Based Access Control model. We discussed the motivations for this research followed by the contribution of this work. The next chapter provides background knowledge that is required for comprehending the rest of the thesis.

Chapter 2

Background

2.1 Introduction

In this chapter, we introduce the concepts used as the basis of this thesis. Since HO(T)-ReBAC is introduced to incorporate temporal dimension in a Relationship-Based Access Control (ReBAC) model, therefore we will present some details about ReBAC in Section 2.2. In HO(T)-ReBAC policies (detailed in Chapter 3), the transient relationships are compared to each other for an authorization decision. These comparisons use the 13 Allen relations. We have provided details about these 13 Allen relations in Section 2.3. Group-Centric Secure Information Sharing Model (g-SIS) has inspired this work, and being able to define g-SIS group operation using HO(T)-ReBAC gave an initial indication for the usefulness of the HO(T)-ReBAC model, which we will explore further in Chapter 4. Section 2.4, will provide details about g-SIS and its group operations. One of the most challenging parts of HO(T)-ReBAC is to efficiently find the periods for the complex composition of transient relationships in HO(T)-ReBAC occurred. We have formalized this problem as a Constraint Satisfaction Problem (CSP); therefore, in Section 2.5, we provide background about CSP and CSP algorithms that are related to this thesis. We will be utilizing these CSP algorithms in Chapter 5.

In the following, we will start by describing the Relationship-Based Access Control Model (ReBAC) model.

2.2 Relationship-Based Access Control Model (ReBAC)

A social network can be represented by an edge-labelled directed graph, G, consisting of a vertex set, V, edge label set, I, and set of edges, E where $E \subseteq V \times V \times I$. The edges of the social network

represent relationships. In Relationship-based access control, the authorization depends on the relationships between the requestor and the resource in a social network. Fong [17] proposed a general-purpose model and also showed the limitations of previous access control models such as RBAC to express the policies that mimic the nature of these complex social computing. Attribute-supporting ReBAC (AReBAC) model proposed by [4] is a combination of Relationship-Based Access Control and Attribute-based Access control [20]. It provides more granular access control in Neo4j [9], which is a graph database. AReBAC proposed a new query language called Nano-Cypher [26] based on Neo4j's Cypher language [5]. Nano-Cypher fused database queries with access control policies making the authorization efficient. AReBAC, more expressive than ReBAC, suffers from the same issues as ReBAC in terms of the temporal attribute.

In AReBAC, Rizvi and Fong [26] model a graph database query as graph homomorphism problem with constraints, where the graph homomorphism problem belongs to a general class of problems known as Constraint Satisfaction Problems. In a graph homomorphism problem given graphs G and H, the question is if there exist a mapping $M : V(G) \rightarrow V(H)$ such that if $(u,v) \in E(G)$ then $(M(u),M(v)) \in E(H)$. The query is defined using graph patterns with added constraints and is executed against the Neo4j graph database. So it checks the existence of the graph patterns that satisfy all the constraints in the graph database. [26] introduced an algorithm named Forward Checking augmented with Live-End Directed Backjumping (*FC-LBJ*) to execute the query efficiently. We will further describe *FC-LBJ* in the following section with other Constraint Satisfaction Problem (CSP) algorithms. HO(T)-ReBAC also involves solving homomorphism problems with different constraints, so we will use *FC-LBJ* as the basis for a new CSP algorithm tailored for constraints in our problem.

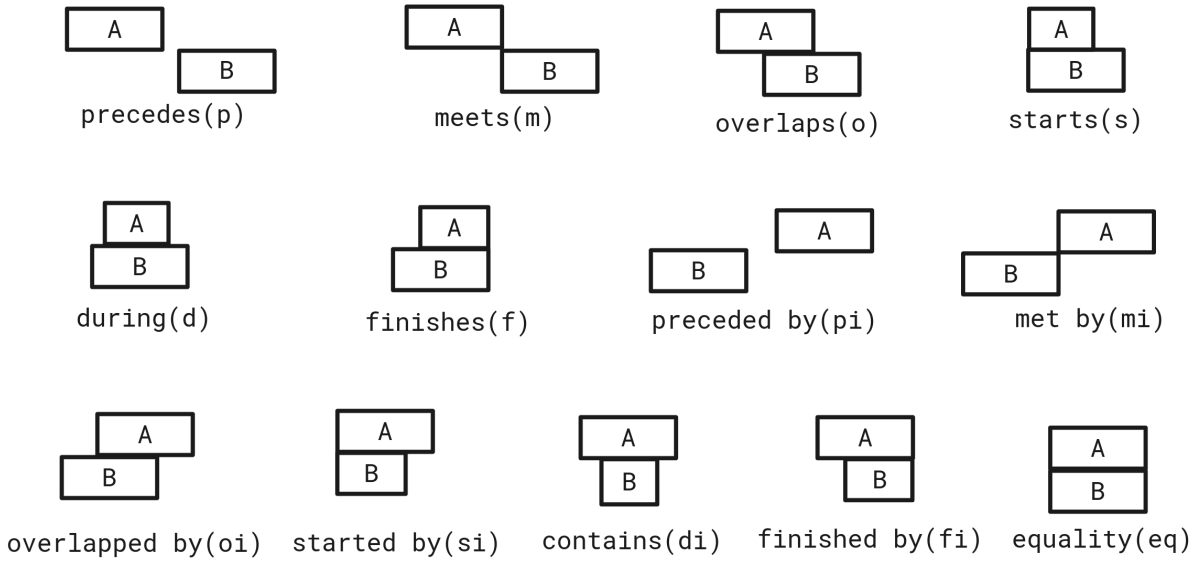
In the next section, we will detail all the 13 Allen relations.

Table 2.1: Allen's 13 basic relations between intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$. This table is recreated from [24].

r	A **r** B	$\mathbf{r}^{-1}$
p	$a_1 < a_2 < b_1 < b_2$	pi
m	$a_1 < a_2 = b_1 < b_2$	mi
o	$a_1 < b_1 < a_2 < b_2$	oi
s	$a_1 = b_1 < a_2 < b_2$	si

r	A **r** B	$\mathbf{r}^{-1}$
d	$b_1 < a_1 < a_2 < b_2$	di
f	$b_1 < a_1 < a_2 = b_2$	fi
eq	$a_1 = b_1 \wedge a_2 = b_2$	eq

Figure 2.1: Allen Relations Visual Representation. This image is recreated from [1].



2.3 Allen relations

Allen [12] identified 13 basic relations between any two time intervals. These relations are known as Allen relations. These relations are exhaustive and distinct. They are distinct because any two intervals are related only to one of the relations; they are exhaustive because all pairs of intervals can be captured by one of the 13 intervals. The universal relation is defined as the set containing all the 13 Allen relations. Given two intervals A and B and an Allen relation r, $A\ r\ B$ describes that A is related to B by the relation r. Now we have disjunctive relation R, a subset of the universal relation; $A\ R\ B$ asserts that there is a r that belongs to R such that $A\ r\ B$. The 13 Allen relations are precedes (p), meets (m), overlaps (o), starts (s), during (d), finishes (f), preceded by (si), met by (mi), overlapped by (oi), started by (si), contains (di), finished by (fi) and equality (eq). Table 2.1 lists all these 13 Allen relations. The relation $\mathbf{r}^{-1}$ represents the converse of $\mathbf{r}$. We have also added a visual representation for the 13 Allen relations in Figure 2.1.

In the following section, we will describe an access control model, namely, Group-Centric Secure Information Sharing (g-SIS), in which a subject can access a resource by being present in the same group as the resource.

2.4 Group-Centric Secure Information Sharing (g-SIS)

Group-Centric Secure Information Sharing (g-SIS) model developed by [23] facilitates the sharing of information among users in the same group. g-SIS provides makes authorization decisions using the memberships of users and objects with a group. The membership of users and objects are defined using the membership properties and membership renewal properties in g-SIS. The following section will provide details about the membership properties and membership renewal properties in g-SIS.

2.4.1 Group Membership in g-SIS

In g-SIS, group membership of users and objects are defined through the membership properties and membership renewal properties. These properties determine how a user can access an object through their membership to a group. A user and an object becomes member of a group using four group operations. These four group operations are:

1. Join: A Join operation indicates a user becoming a member of a group.
2. Add: An Add operation indicates an object becoming a member of a group.
3. Leave: A Join operation indicates a user ceases to be a member of a group.
4. Remove: An Add operation indicates an object ceases to be a member of a group.

Before we present the group membership properties and membership renewal properties, we need to know two core properties of g-SIS. These two core properties are:

1. *Authorization Persistence*: If a user, u, has access to object o, then u remains authorized to access o until a group operation occurs that revokes the access.
2. *Revocation Persistence*: If a user, u, does not have access to object o, then u remains unauthorized to access o until a group operation occurs that gives the access.

Membership properties: The membership properties in g-SIS are defined using 4 groups operations: Join, Add, Leave, and Remove. Join and Leave operations are used for users, whereas Add and Remove operations are used for objects. Using Join and Add operations, a user and an object become part of a group. Using Leave and Remove, a user and an object cease to be part of a group. In g-SIS, there are two variants (i.e., Strict and Liberal) for the 4 group operations. The Strict and Liberal variants of these operations are related to restrictiveness introduced for the access.

The Strict variant of group operations is more restrictive than the Liberal variant. In Strict Join and Strict Add, a user can access an object in a group only if the user joined the group before the

object was added to the group. Whereas in Liberal Join and Liberal Add, a user can access an object in a group irrespective of whether the user joined the group before or after the object was added. In Strict Leave and Strict Remove, a user loses access to an object when the user or the object is no longer a group member. In Liberal Leave and Liberal Remove, a user retains access to an object in a group even when the user or the object is no longer a group member. Chapter 4 will provide further details on how a user can access an object using the membership properties. We will also discuss how to emulate these properties using HO(T)-ReBAC policies.

Membership renewal properties: We have seen details about membership properties. Let us now explore more details about membership renewal properties. Membership renewal properties deal with access when a user rejoins and subsequently leaves the group. There are 6 membership renewal properties. These are Lossless Join, Lossy Join, Nonrestorative Join, Restorative Join, Gainless Leave, and Gainful Leave. Lossless and Lossy Join are about how a user loses authorizations held prior to joining a group. Nonrestorative and Restorative Join are about restoring past authorization held by a user upon joining a group. Gainless and Gainful Leave are about gaining authorizations when a user leaves a group. Chapter 4 will provide further details about these membership renewal properties. Later we will also discuss how to emulate these properties using HO(T)-ReBAC policies.

In the following section, we will describe the Constraint Satisfaction Problem (CSP) and some algorithms for solving CSP to find periods of materialization for the complex composition of transient relationships.

2.5 Constraint Satisfaction Problem Algorithms

A Constraint Satisfaction Problem (CSP) is a problem requiring its solutions within in some limitations. These limitations are known as constraints. A CSP as defined by [25] consists of a set of variables $V = \{V_1, V_2, V_3, \ldots, V_n\}$ and for each variable $V_i \in V$ there exits a finite domain $D_i = \{v_{i1}, v_{i2}, v_{i3}, v_{i4}, \ldots, v_{im}\}$. Variable V_i may be assigned a value $v_{ij} \in D_i$. There is also a set of

constraints $C = \{C_1, C_2, C_3, \ldots, C_k\}$. In our case we are only considering binary constraint so set C can be represented as $C = \{C_{11}, C_{12}, C_{13}, \ldots, C_{1n}, C_{21}, C_{22}, C_{23}, \ldots, C_{2n}, \ldots, C_{n1}, C_{n2}, C_{n3}, \ldots, C_{nn}\}$, where constraint C_{ij} is a relation between V_i and V_j and if C_{ij} is null then that would mean no constrain exists between V_i and V_j. Solving an instance of a CSP means to find a value assignment for each variable such that such that all constraints are satisfied.

One of the variants of the CSP is the graph homomorphism problem. As described in Section 2.2, in a graph homomorphism problem given graphs G and H, the question is if there exist a mapping $M : V(G) \to V(H)$ such that if $(u,v) \in E(G)$ then $(M(u), M(v)) \in E(H)$. In this thesis, we will be dealing with the graph homomorphism problem but with additional constraints, that is why an understanding of CSP is vital for the thesis. In the following sections we will describe some of the algorithms that are use to solve CSP. Section 2.5.1 describes the Conflict-Directed Backjumping algorithm that is one of the well-know backtracking algorithms for solving CSP. Section 2.5.2 describes the Forward Checking algorithm that is one of the well-known lookup algorithms for solving CSP. Section 2.5.3 describes a new algorithm named *Forward Checking augmented with Live-End Directed Backjumping (FC-LBJ)* [26] that combines backtracking and lookup part for solving graph homomorphism with constraints. *FC-LBJ* reduces constraint checking when we are gathering all solutions rather than finding just one solution. Detailed explanation for the algorithms presented in Section 2.5.1 and Section 2.5.2 can be found in [25], and the detailed explanation of algorithm presented in Section 2.5.3 can be found in [26]. We will be giving a high-level explanation for all these algorithms in the following sections.

2.5.1 Conflict-Directed Backjumping

Conflict-Directed Backjumping (CBJ) is a backtracking algorithm that is used in solving CSP. As the name suggests, *CBJ* is a related backjumping (BJ) algorithm. In BJ, when the current variable, v_c, reaches a dead end in the search tree, then the algorithm jumps back to the variable, v_n, that conflicted with the v_c, and if the domain of this conflicted variable does not have more values to check then the algorithm chronologically backtracks. However, in the case of *CBJ*, when v_c

Algorithm 1: csp-procedure

```
   Input: n and consistent
1  consistent ← true;
2  status ← "unknown";
3  i ← 1;
4  while status = "unknown" do
5  |  if consistent then
6  |  |  i ← label(i, consistent);
7  |  else
8  |  |  i ← unlabel(i, consistent);
9  |  if i > n then
10 |  |  status ← "solution";
11 |  else if i = 0 then
12 |  |  status ← "impossible";
```

Algorithm 2: cbj-label

```
   Input: i and consistent
1  consistent ← false;
2  for v[i] ← EACH ELEMENT OF current-domain[i] while not consistent do
3  |  consistent[j] ← true;
4  |  for h ← 1 TO i − 1 while consistent do
5  |  |  consistent ← check(i, h);
6  |  if not consistent then
7  |  |  pushnew(h − 1, conf-set[i]);
8  |  |  current-domain[i] ← remove(v[i], current-domain[i]);
9  if consistent then
10 |  return i + 1;
11 else
12 |  return i;
```

Algorithm 3: cbj-unlabel

```
   Input: i and consistent
1  h ← max-list(conf-set[i]);
2  conf-set[h] ← remove(h, union(conf-set[h], conf-set[i]));
3  for j ← h + 1 TO i do
4  |  conf-set[j] ← {0};
5  |  current-domain[j] ← domain[j];
6  current-domain[j] ← remove(v[h], current-domain[h]);
7  consistent ← current-domain[h] ≠ nil;
8  return h;
```

reaches a dead end in the search tree, then the algorithm jumps back to v_n, and if the domain of this conflicted variable does not have more values to check then, the algorithm jumps back to following most recent variable that conflicted with either v_c or v_n.

Algorithm 1, Algorithm 2 and Algorithm 3 give the pseudo-code for the *CBJ* algorithm. This pseudo-code was presented in [25]. Algorithm 1 for given set of conflicts between variables, finds the first instantiation of the variables that satisfies all the conflicts. The function *label* in Algorithm 1 tries to find the consistent instantiation for the current variable and return the value for the next variable. The function *unlabel* in Algorithm 1 is use to perform backtracking when the algorithm reaches a dead end. For *CBJ*, *fc-label* (Algorithm 2) and *cbj-unlabel* (Algorithm 3) replaces the functions *label* and *unlabel* functions respectively. In *CBJ*, we maintain a globally declared conflict set for every variable that contains the variables that failed the consistency check with the given variable. So we can use this conflict set to backtrack to the deepest variable in the search tree that is present in the conflict set of the current variable whenever we reach a dead-end for the current variable. This tracking of conflicts improves the efficiency of *CBJ* compared to chronological backtracking and backjumping algorithms.

2.5.2 Forward Checking

Algorithm 4: fc-label

```
   Input: i and consistent
 1 consistent ← false;
 2 for v[i] ← EACH ELEMENT OF current-domain[i] while not consistent do
 3 |   consistent ← true;
 4 |   for j ← i+1 TO n while consistent do
 5 |   |   consistent ← check-forward(i, j);
 6 |   if not consistent then
 7 |   |   pushnew(h−1, conf-set[i]);
 8 |   |   current-domain[i] ← remove(v[i], current-domain[i]);
 9 |   |   undo-reduction(i);
10 if consistent then
11 |   return i+1;
12 else
13 |   return i;
```

Algorithm 5: fc-unlabel

Input: i and *consistent*

```
1 h ← i − 1;
2 undo-reduction(h);
3 update-current-domain[i];
4 current-domain[i] ← remove(v[h], current-domain[h]);
5 consistent ← current-domain[h] ≠ nil;
6 return h;
```

Forward checking (FC) is a look-ahead technique used in CSP that check whether the current variable assignment can be extended to future values by checking the domain of future variable assignment. Forward checking also removes all the candidates from the domain of future variables that are inconsistent with the current assignment. The current and the past assignments are known as the partial solution. Forward checking does not concerns with the backtracking or backjumping but it make the search efficient by reducing the search space which is done through the removal of inconsistent future values with the current assignment. *FC* has been augmented with backtracking algorithms such has *CBJ* to improve the efficiency of the overall algorithm. [26] also augmented *FC* with live-end backjumping (*LBJ*) algorithm where *LBJ* is another backtracking algorithm developed by [26] that builds on *CBJ*.

Algorithm 1, Algorithm 4 and Algorithm 5 gives the pseudo-code for the *FC* algorithm. This pseudo-code was presented in [25]. We have already described Algorithm 1 along with function *label* and *unlabel* in Section 2.5.1. In *FC*, *fc-label* (Algorithm 4) and *fc-unlabel* (Algorithm 5) replaces the functions *label* and *unlabel* functions respectively. The function *fc-label* is used to filter the domain of future variables so that the domain consists of values consistent with the instantiation of the current variable. This reduction in the domain size reduces the search space, thus improving the search efficiency. The function *fc-unlabel* performs chronological backtracking once the search space reaches a dead end. So this gives FC the potential to be paired with efficient backtracking algorithms to improve the search efficiency that we get by using FC alone.

2.5.3 Forward Checking augmented with Live-End Directed Backjumping Algorithm

Algorithm 6: GPEvalInit

Input: The graph pattern: GP
Input: Context information: $Info : CA(c) \rightarrow V(G')$
Global: Attribute-supporting edge-labeled and directed graph: (G', A_V', A_E')
Global: The result set RS is a finite set of functions, each having the function signature $Ret \rightarrow V(G')$
Local: The candidate sets for the vertices: $Cand : V(G) \rightarrow 2^{V(G')}$

```
1 Initialize RS and Cand to ∅;
2 foreach v ∈ Dom(Info) do
3 |   Cand(v) ← {Info(v)};
4 if PreCheck {GP, Cand} then
5 |   GPEvalRec (GP, Cand, ∅, ∅);
6 |   return RS;
7 else
8 |   return ∅;
```

Forward Checking algorithm [25], augmented with Live-End Directed Backjumping, was proposed by Rizvi and Fong in [26]. As the name suggests, the algorithm combines live-end backjumping with forward checking, where live-end backjumping forms the backtracking part and forward checking, as previously discussed, forms the lookup part of the algorithm. The live-end backjumping builds upon the Conflict-Directed Backjumping algorithm. *LBJ* backtracks similar to *CBJ* when the algorithm encounters a dead end. The difference happens once the algorithm finds a search result. *LBJ* works for scenarios where there are multiple search results, but if the algorithms stop as soon as it finds a result, then in those cases, *LBJ* will behave the same as *CBJ*. In *CBJ*, the algorithm chronologically backtracks once it finds a search result, but in *LBJ*, instead of chronological backtracking, the algorithm backjumps to find a result belonging to a different equivalence class than the most recent result.

Algorithm 6, Algorithm 7 and Algorithm 8 gives the pseudo-code for *CBJ* algorithm. This pseudo-code was presented in [26]. Algorithm 6 represents the initialization phase of the *FC-LBJ* algorithm. In this step, initialization of the global and local variables is carried, and initial constraints checks are performed. Algorithm 7 performs the backtracking part of the algorithm.

Algorithm 7: GPEvalRec

Input: GP and $Cand$ as specified in Algorithm 6
Input: The vertex assignment: $Assn : V(G) \to V(G')$
Input: The incoming conflict sets for the vertices: $ConfIn : V(G) \to 2^{V(G)}$
Local: A candidate sets for the vertices: $Cand' : V(G) \to 2^{V(G')}$
Local: A vertex assignment: $Assn' : V(G) \to V(G')$
Local: The set of vertices for backjumping: $Conflicts \subseteq 2^{V(G)}$
Local: The set of outgoing conflicts: $ConfOut \subseteq 2^{V(G)}$
Local: The current target vertex: $v \in V(G)$

if $|V(GP)| = |Dom(Assn)|$ **then**
    $RS \leftarrow RS \cup Assn\,|_{Ret}$;
    return Ret;
$deadEnd \leftarrow \top$;
$ConfOut \leftarrow \emptyset$;
$Conflicts \leftarrow \emptyset$;
$v \leftarrow$ **PickNextVertex** $(GP, Assn, Cand)$;
MexFilter $(GP, Assn, Cand(v), ConfIn)$;
forall $n \in Cand(v)$ **do**
    $Cand' \leftarrow Cand \setminus \{v\}$;
    $Assn' \leftarrow Assn \cup \{v \mapsto n\}$;
    $ConfIn' \leftarrow ConfIn$;
    $valid \leftarrow$ **FC** $(GP, Assn', Cand', v, ConfIn', ConfOut)$;
    if $valid$ **then**
        $deadEnd \leftarrow \bot$;
        $Jump \leftarrow$ **GPEvalRec** $(GP, Cand', Assn', ConfIn)$;
        if $Jump \neq \emptyset \wedge v \notin Jump$ **then**
            return $Jump$;
        else
            $Conflicts \leftarrow Conflicts \cup Jump$;
if $(deadEnd)$ **then**
    $Conflicts \leftarrow Conflicts \cup ConfIn(v) \bigcup_{u \in ConfOut} ConfIn(u)$;
    return $Conflicts$;
else
    $Conflicts \leftarrow Conflicts \cup Ret \bigcup_{u \in Ret} ConfIn(u)$;
    if $ConfOut \neq \emptyset$ **then**
        $Conflicts \leftarrow Conflicts \cup ConfIn(v) \bigcup_{u \in ConfOut} ConfIn(u)$;
    return $Conflicts$;

Algorithm 8: GPEvalFC

Input: *GP* and *Cand* as specified in Algorithm 6
Input: $Assn, v, ConfIn$, and *ConfOut* as specified in Algorithm 7
Local: The set of relevant edges: $Edges \subseteq 2^{E(G)}$
Local: The result of neighbourhood retrieval query: $neighbours \subseteq 2^{V(G')}$
Local: The other vertex from an edge: $u \in V(G)$

```
37 Edges = getRelevantEdges(G, v);
38 foreach e ∈ Edges do
39     u = getOtherVertex{e, v};
40     if u ∉ Dom(Assn) then
41         neighbours = getNeighbours(GP, e, v, Assn(v));
42         if (Cand(u) = ∅) ∨ (Cand(u) ⊈ neighbours) then
43             ConfIn(u) ← ConfIn(u) ∪ ConfIn(v) ∪ {v};
44         if Cand(u) = ∅ then
45             Cand(u) ← neighbours;
46         else
47             Cand(u) ← Cand(u) ∩ neighbours;
48         if Cand(u) = ∅ then
49             ConfOut ← ConfOut ∪ {u};
50             return ⊥;
51 return ⊤;
```

It implements the live-end backjumping. Live-end backjumping extends Conflict-Directed Backjumping algorithm when finding multiple search results is required. Algorithm 8 performs the lookup using forward checking in *FC-LBJ* . The forward checking step in *FC-LBJ* also involves populating the values for the domain for the future variables. It then filters the domain for the future variables by removing the value that is not consistent with the instantiation of the current variable. The next step is validating whether, after filtering, any values remain in the domain. If the domain becomes empty, that means a dead end has been reached, and the algorithm has to backtrack.

This chapter described concepts that help to support this research. The next chapter will present a formalization for the HO(T)-ReBAC model. We will also present a formal language for specifying policies in HO(T)-ReBAC.

Chapter 3

HO(T)-ReBAC

3.1 Introduction

Technologies such as the Internet of Things (IoT) have increased the interactions among users and resources such as IoT devices. Many of these interactions are transient, meaning these interactions begins at some point in time and end at some other time point and begins again and ends again. A person's presence in a smart home can be seen as an interaction between the person and the smart home. Every time a person enters the smart home, an interaction begins, and it ends when the person leaves the smart home. These interactions can be viewed as *transient relationships* among users and resources. Every interaction corresponding to one transient relationship. We can use the *Relationship-Based Access Control (ReBAC)* model to model access control in these scenarios. In ReBAC, access control policies describe how a user and a resource are related to one another by complex compositions of relationships. The issue in using the ReBAC model is that in ReBAC, we can not use the historical transient relationships and only have to rely on the transient relationships that have begun and have not ended. Therefore, the model can only track the current transient relationships and can not record past transient relationships. Not being able to utilize the history of transient relationships motivated us to come up with a new access control model called *Higher-Order (Temporal) Relationship-Based Access Control (HO(T)-ReBAC)* model to introduce the ability to use the history of transient relationships for an authorization decision. By incorporating the history of transient relationships, we introduced a temporal dimension to the ReBAC model. With HO(T)-ReBAC, we presented the following contributions in detail in this chapter.

1. Introduced a way to facilitate the enforcement of policies that are condition on

transient relationships.

2. Presented a way to efficiently find time periods for the materialization of the complex composition of transient relationships.

3. Presented a policy language in which relations between the time periods of materialization determine the authorization decision.

Figure 3.1: HO(T)-ReBAC Components

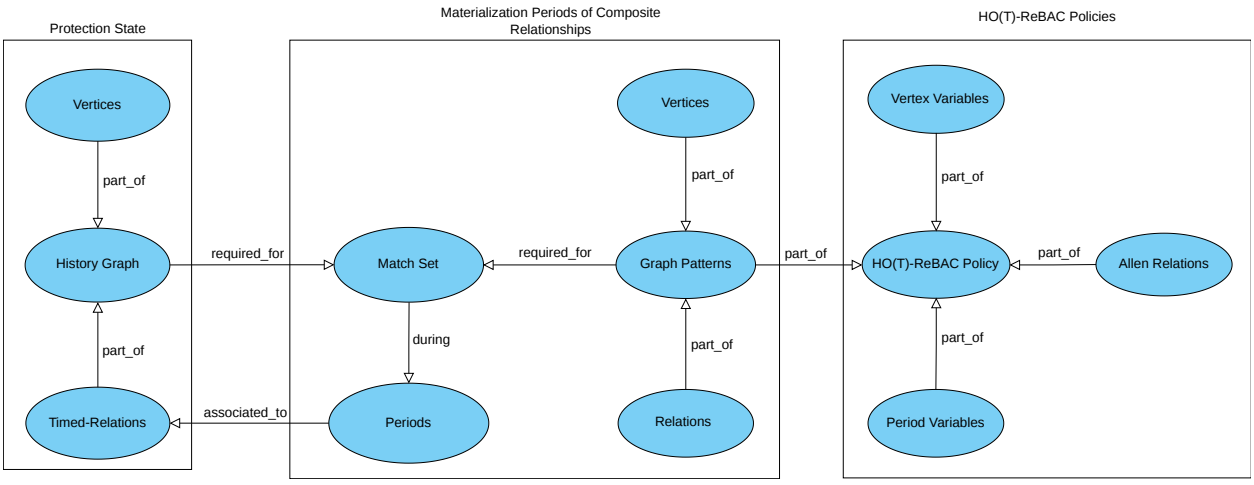


Figure 3.1 shows the components for HO(T)-ReBAC model. The Figure is divided into three parts, where each part shows one of the three contributions described above. The leftmost box is the protection state box. A protection state is a collection of information tracked by the access control system in order for the later to make authorization decision. It is the snapshot of the system, describing the aspects of the system that are relevant to access control. The *Protection State* box shows the components for the first contribution where we have a *history graph*, which is a graph to track all the users and resources as well as the transient relationships between them. Section 3.2 will detail the protection state of HO(T)-ReBAC.

In Figure 3.1 *Materialization Periods of Composite Relationships* box shows the components for the second contribution. One motivation behind HO(T)-ReBAC is to handle complex relationships between entities such as users, resources, and groups efficiently. Complex relationships

occur between two entities when these two entities are not directly connected through a relationship but through some collection of vertices and relationships. Since this complex relationship is made using multiple relationships and vertices, we call it *complex composition of relationships*. By finding the period of materialization of a complex composition of relationships, we are essentially finding the periods during which this complex composition of relationships was active. In ReBAC, a policy describes how a user and a resource are related to one another by complex compositions of relationships. Such a policy is satisfied or not essentially amounts to a graph matching problem. Now we are introducing a temporal dimension to relationships and checking a complex composition of relationships over time. Just like transient relationships are active during some time period, we say a complex composition of relationships is active when it materializes. To materialize a complex composition of relationships during a time period, all relationships that make up the complex composition need to be active during that time period. Therefore materialization of the complex composition of relationships means all relationships in it are active. In the past, an attempt has been made to introduce a temporal dimension to ReBAC [16]. The experience was that the performance was demanding and required changing of overall graph structure whenever there is a change. Our work allows efficient computation of time periods of materialization of these composite relations. *Graph patterns* are the graph structures that are used in HO(T)-ReBAC to describe the complex composition of relationships and to find a *Match*, we can find the solution for the graph matching problem between a graph pattern and history graph to find the time periods of materialization of these complex composition of relationships without changing the structure of the history graph. We will give details of algorithms to find the materialization period in a later chapter. This chapter will give a mathematical formalization of these materializing time periods in Section 3.3.

The third contribution relates to the access control policies in HO(T)-ReBAC and is shown in the *HO(T)-ReBAC Policies* box in Figure 3.1. Once we have found the periods of materialization for the complex composition of relationships, we need a way to use them for authorization

decisions. Therefore, we need a language that will use these periods of materialization to define authorization decision policies in HO(T)-ReBAC. Hence, HO(T)-ReBAC access control policies utilize and incorporate relationships' transient nature to make authorization decisions. These authorization policies will keep the system secure by ensuring authorization is given when a requestor fulfills all requirements in the policy. HO(T)-ReBAC access policies rely on how the materialization time periods are related to each other, and we use the 13 Allen relations [12] to check the relations between the materialization time periods. Section 3.4 will describe the syntax and semantics of the HO(T)-ReBAC policies.

3.2 HO(T)-ReBAC Protection State

In access control, as described earlier a protection state is a collection of information tracked by the access control system to make authorization decision. In ReBAC, a protection state tracks *relationships* between users and resources but in HO(T)-ReBAC, we are concerned with *transient relationships*; therefore, the protection state in HO(T)-ReBAC tracks transient relationships between entities such as users, resources or groups. A transient relationship in HO(T)-ReBAC occurs during a period of time, whereas a relationship in ReBAC has no period associated with it. A transient relationship in HO(T)-ReBAC is active during a time period for which it occurs and inactive during the rest of the periods. Whereas a relationship does not have active or inactive periods, a relationship in ReBAC either exists or does not exist. Before we formally define a history graph, we need to define time periods associated with transient relationships because the presence of these time periods that allow transient relationships to be transient. We refer to these time periods as *Periods*. These periods must have a beginning, but some of these periods may have an ending, and some may not: such a period is still ongoing.

Definition 3.2.1 (Period) *A closed period* $[x,y]$ *is the set* $\{z \in R | x \leq z \wedge z \leq y\}$. *An ongoing period* $[x,\infty]$ *is the set* $\{z \in R | x \leq z\}$. *We define* P_{closed} *as the set of all legitimate closed periods and* P_∞ *as the set of all legitimate ongoing periods.* $\mathscr{P}$ *is defined as the set of all legitimate closed and*

legitimate ongoing periods (i.e., $\mathscr{P} = P_{closed} \cup P_{\infty}$).

An ongoing period $[x, \infty]$, as the name suggests, is a period that is still taking place. This means that this period has not ended yet, and therefore, the end time of the period is not known yet. Since we do not know the end time of an ongoing period, we use a special symbol ∞ as a placeholder for the end time. This special symbol ∞ does not mean that the period will last forever; it is just a placeholder to signify that the end time of the period is yet unknown. To emphasize, an ongoing period is not a period that will never end but a period that has not ended yet.

Definition 3.2.2 (History Graph) *A history graph H is a three tuple (V, L, F) consisting of a set V of vertices, a set L of relation identifiers, and a set $F \subseteq V \times V \times L \times \mathscr{P}$ of directed relations such that for every pairs of timed-relationships (u, v, l, P) and $(u, v, l, P') \in F$ where $u, v \in V$, and $l \in L$, the periods P and P' are either equal $(P = P')$ or disjoint $(P \cap P' = \varnothing)$.*

A history graphs consist of vertices and timed-relationships. A single vertex in the history graph uniquely represents a user, group, or resource. A timed-relationship describes a single transient relationship between any two vertices in the history graph, and a period describes the time during which the transient relationships occurred. A period is an attribute associated with the timed-relationship. A timed-relationship is said to be active during a period that it is associated with.

Two types of periods can be associated with a timed-relationship (i) closed periods and (ii) ongoing periods. When a timed-relationship is associated with a closed period, it means that the transient relationship represented by that timed-relationship is no longer active. The start time of the closed period tells when that transient relationship became active, and the end time of the closed period tells when that transient relationship terminated. Whereas when a timed-relationship is associated with an ongoing period, it means that the transient relationship represented by that timed-relationship is still active. The start time of the ongoing period tells when that transient relationship became active, and as mentioned, this transient relationship has not terminated yet, so

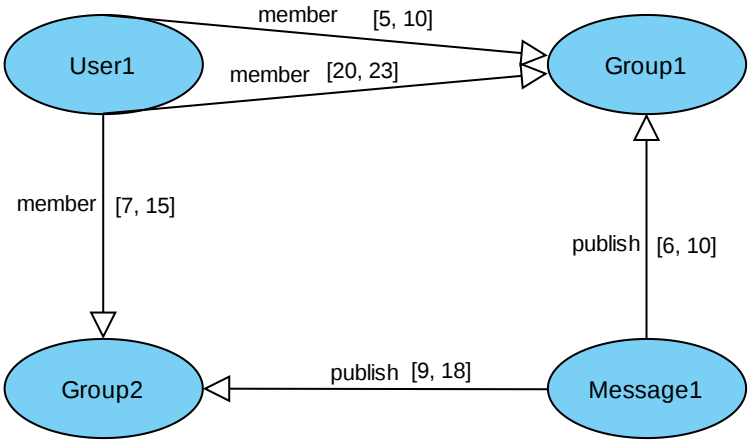
there is no end time. We use the special symbol ∞ as a placeholder for the end time when the end time is not known yet. It does not mean that this transient relationship is permanent, it just means it has not terminated yet. When a transient relationship is activated, a timed-relationship is added to the history graph with an ongoing period. When this transient relationship is terminated, the period associated with the timed-relationship is revised to a closed period since we now know the end time of the period.

Since a timed-relationship represents only one transient relationship, we can have multiple timed-relationships between any two vertices in the history graph. Now, since a timed-relationship represents a transient relationship, it does not make sense for a new transient relationship to begin between some pairs of vertices in the history graph while there is still some other transient relationship going on between the same pair of vertices in the history graph. Therefore, any two timed-relationships with same relation identifier between the same pair of vertices in the history graph can not have overlapping periods associated with them to ensure this behaviour. That is why distinct periods P and P' must be disjoint in Definition 3.2.2.

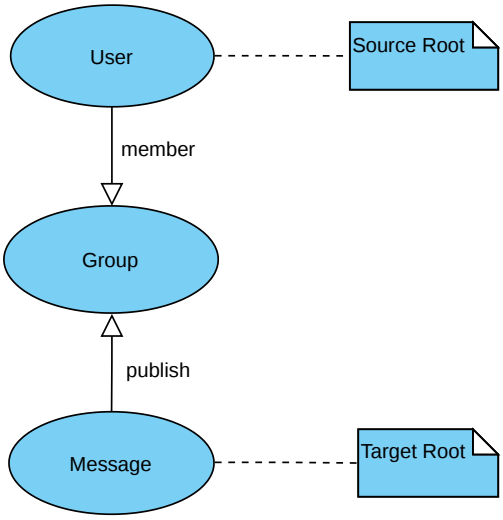
Example 3.2.1 *Consider an example of an online messaging service like Whatsapp [11], which provides a group chat feature. A typical group messaging service will have three types of vertices: users, groups, and messages. Users belonging to the same group can share messages. In Figure 3.2, the history graph contains vertices users, groups, and messages where users are represented by* USER1*,* USER2*, and messages are represented by* MESSAGE1*,* MESSAGE2 *and groups are represented by* GROUP1*. There are five timed-relationships, three with relation identifier* MEMBER *between a user and a group and two with relation identifier* PUBLISH *between a message and a group. Every timed-relationship between any two vertices in the history graph has a period and a relation identifier associated with it. The associated period will determine the time for which that timed-relationship was active. In Figure 3.2, the timed-relationship* MEMBER *have periods* $[5,10]$*,* $[7,15]$ *associated with them and timed-relationship* PUBLISH *have periods* $[6,10]$*,* $[9,18]$*, associated with them. The associated periods with two timed-relationships between* USER1 *and*

Figure 3.2: HO(T)-ReBAC Components for Whatsapp Use Case

(a) History Graph



(b) Graph Pattern



GROUP1 *described the two distinct periods when* USER1 *belonged to* GROUP1. *We will use this use case of Whatsapp in later sections to describe other components of HO(T)-ReBAC and will refer it to as the* Whatsapp Use Case.

This section described the protection state in HO(T)-ReBAC that captures all the users, groups, and resources and their transient relationships in a graph structure called the history graph. We also saw the periods associated with these transient relationships and how the transient relationships are said to be active. The following section will show how we can define the complex composition of relationships between two vertices using graph structures and check for the satisfaction of those composite relationships in a history graph. We will also see how we define and find the periods of materialization for these composite relationships.

3.3 Materialization Periods of Composite Relationships

This section will describe the materialization of composite relationships and the periods associated with the materialization. In Section 3.3.1, we will describe what we mean by the complex composition of relationships. We will refer to these complex compositions of relationships as *composite relationships* for the rest of the thesis for brevity. Section 3.3.2 will describe when these composite relationships are satisfied, and Section 3.3.3 and Section 3.3.4 will give insights on the periods associated with these composite relationships.

3.3.1 Graph Pattern

A graph pattern describes a relation that may hold between some pairs of vertices in the history graph. A graph pattern is intended to be used for specifying authorization conditions. That is, a graph pattern typically specifies how a resource requestor shall be related to the requested resource in order for access to be granted. Before we talk about the composite relationships' materialization, we need to know how the composite relationships are formed in HO(T)-ReBAC. We use a graph structure that allows us to define relations between vertices that are not be directly related to each

other but through other vertices and relationships. In this graph structure, we distinguish the two vertices between which the composite relationship is being defined. We refer to this graph structure as a graph pattern. Definition 3.3.1 defines a graph pattern formally.

Definition 3.3.1 (Graph Pattern) *A graph pattern GP is a five tuple* (U, L, E, u, v)*, consisting of a set* U *of vertices, a set* L *of relation identifiers, a set* $E \subseteq U \times U \times L$ *of directed edges, and two vertices* $u, v \in U$*. Graph*(U, L) *defines the set of all graph patterns with vertices coming from the countably infinite set* U*, and relation identifiers coming from the finite set* L*.*

Since we needed a graph structure with two distinguished vertices, we opted for birooted graphs to define graph patterns. A birooted graph is simply a graph G with two vertices that belong to G. We use the two distinguish vertices u and v of the graph pattern to represent the vertices in the graph pattern between which the composite relationships are being defined. We will use *relationships* for the directed edges of graph patterns. Graph patterns are used to inform what relationships and vertices in the graph pattern form a composite relationship. Now that we have two graph structures (i.e., history graph and graph pattern) so we would like to reiterate the naming convention that we will follow throughout the thesis. A history graph consists of vertices and timed-relationships, whereas a graph pattern consists of vertices and relationships. The vertices in the history graph represent actual users, groups, or resources, whereas vertices in the graph pattern are placeholders for vertices in history graph. For example few vertices in the history graph for users can be USER1, USER2, USER3. Also, timed-relationships represent actual transient relationships, whereas the relationships are placeholder for the transient relationships. A timed-relationship in a history graph has a period associated with it because a timed-relationship corresponds to an actual transient relationship. On the other hand, a relationship in a graph pattern is a placeholder for a timed-relationship, so it does not have a period associated with it.

Example 3.3.1 *In continuation of the* Whatsapp Use Case*, in Figure 3.2 (b) we can see a graph pattern to describe a composite relationship between a user and message. The graph pattern*

consists of three vertices USER, GROUP, *and* MESSAGE, *two relationships with relation identifier* MEMBER *and* PUBLISH *and is birooted at vertices* USER *and* MESSAGE. *The vertices in the graph pattern* USER, GROUP, *and* MESSAGE *represents user, group, and message respectively. The graph pattern is an authorization condition for a user to read a message meaning the permission is given to a user to read a message only when the user and the message are related to one another through the given graph pattern. The given graph pattern requires a group's existence such that both the user and the message are related to it with relation identifier* MEMBER *and* PUBLISH *respectively.*

3.3.2 Match

A *match* is used to find the composite relationship defined using a graph pattern in a history graph. A match finds the solution for the graph matching problem where the vertices and relationships of the given graph pattern are mapped to vertices and timed-relationships of the given history graph with temporal constraints. For a composite relationship to be active during a period, all the mapped periods associated with timed-relationships must be active during that period. Therefore the time constraint we are interested in is the existence of a period when all timed-relationships were active. Also, for finding the composite relationship, we fix the two root vertices of the graph pattern to two vertices in the history graph. Definition 3.3.2 formally defines a match.

Definition 3.3.2 (Match) *Suppose we are given a history graph $H = (V, L, F)$, a graph pattern $GP = (U, L, E, u_1, u_2)$, and vertices in the history graph $v_1, v_2 \in V$, and a period $P \in \mathscr{P}$, then $\pi : U \to V$ is a Match during P if and only if $\pi(u_1) = v_1$, $\pi(u_2) = v_2$, and for every relationship $(u_3, u_4, l) \in E$, there is a timed-relationship $(\pi(u_3), \pi(u_4), l, P') \in F$ such that $P \subseteq P'$.*

We require a graph pattern, a history graph, and two vertices in the history graph to find a match. A match ensures that the graph pattern's two roots are mapped to the two vertices in the history graph for which the composite relationship is being found. A match during a period P means that all the mapped timed-relationships were active during period P.

Example 3.3.2 *In continuation of the* Whatsapp Use Case*, to find the composite relationship described by the graph pattern, we need to find a match in the history graph. The graph pattern is birooted at* USER *and* MESSAGE*, and we intend graph pattern vertex* USER *to be mapped to history graph vertex* USER1 *and graph pattern vertex* MESSAGE *to be mapped to history graph vertex* MESSAGE1.

In Figure 3.2, there are two matches for the graph pattern in the history graph. First because USER1 *and* MESSAGE1 *have overlapping timed-relationships with* GROUP1 *during the period* $[6,10]$. *We have to note that the period* $[6,10]$ *is the maximal period for the match. The match is found for all periods that are contained inside the period* $[6,10]$. *The second match is found because* USER1 *and* MESSAGE1 *have overlapping timed-relationships with* GROUP2 *during the period* $[9,15]$. *Again,* $[9,15]$ *is the maximal period for the second match, and a match can be found for all periods contained in the period* $[9,15]$.

The two maximal periods found (i.e., $[6,10]$ *and* $[9,15]$*) also have an overlapping period* $[9,10]$*, but that does not need to be the case. The periods found when a graph pattern is matched to a history graph can be either (i) disjoint, (ii) overlap with each other, or (iii) one contains the other.*

3.3.3 Discoverable Period

As we have seen in Section 3.3.2, when we find a match for a graph pattern against a history graph, then the mapped timed-relationships must have a common period. The match is said to be during that common period. But it can be argued that if a match is during a period P, then it is also during periods that are contained in P. However, those periods do not give the entire materialization period; hence we must focus on the maximal period during which a match happens. We refer to this maximal period for a match as a *discoverable period*. Definition 3.3.3 formally defines a discoverable period.

Definition 3.3.3 (Discoverable Period) *Given a history graph* $H = (V,L,F)$*, and a graph pattern* $GP = (U,L,E,u_1,u_2)$*, and two vertices in the history graph* $v_1, v_2 \in V$*, a period P is a discoverable*

period if and only if:

1. *there exists a match π during the period P, and*
2. *there does not exist another period P' for which $P \subset P'$ and π is a match during P'*

We write discoverable(H, u, v, GP) to denote the set of all discoverable periods for history graph H, graph pattern GP, and $u, v \in V$.

When we match a graph pattern against a history graph, we map each vertex and relationship from the graph pattern to the vertices and timed-relationships of the history graph, respectively. The constraint in the mapping is that there must be some period during which the mapped timed-relationships were active. The maximal period during which all mapped timed-relationships were active is called the discoverable period. When a graph pattern is matched against a history graph, it is possible to find more than one match because the set of mapped vertices and timed-relationships in the history graph are different. This would lead us to find multiple discoverable periods, and all of these periods are part of the discoverable periods set.

Example 3.3.3 *In* Whatsapp Use Case*, when we try to find the composite relationship between a user and a message as described by the graph pattern in Figure 3.2 in the history graph in Figure 3.2, we get two discoverable periods* $[6, 10]$ *and* $[9, 15]$. *The first period (i.e.,* $[6, 10]$*) comes from the overlapping period between timed-relationship* MEMBER *between* USER1 *and* GROUP1 *with period* $[5, 10]$ *and timed-relationship* PUBLISH *between* MESSAGE1 *and* GROUP1 *with period* $[6, 10]$. *The second period (i.e.,* $[9, 15]$*) comes from the overlapping period between timed-relationship* MEMBER *between* USER1 *and* GROUP2 *with period* $[7, 15]$ *and timed-relationship* PUBLISH *between* MESSAGE1 *and* GROUP2 *with period* $[9, 18]$.

3.3.4 Official Period

The motivation to introduce the *official period* is to define what period of materialization is for the complex composite relationships. As we have described earlier, a period of materialization for

the complex composition of relationships tells us when this complex composition of relationships was active. For example, for a user and a group, a period of materialization is when the user and the group are related through some complex composition of relationships. We need these periods materialization complex to make authorization decisions because in HO(T)-ReBAC, the policies make the authorization decision based on these periods of materialization. In the next section, we will give further details on how these periods of materialization (i.e., official periods) are used by HO(T)-ReBAC policies.

We have seen how we can find periods for which there is a match between the graph pattern and the history. This section will describe how we can use those periods, i.e., discoverable periods, to find materialization periods. As seen in Example 3.3.2 and Example 3.3.3, when a graph pattern is matched against a history graph, then we may get more than one discoverable period. These discoverable periods can be either (i) disjoint, (ii) overlap with each other, or (iii) one contains the other. In case the periods are disjoint, then all periods are the periods of materialization for the composite relationship. If one period contains the other, then the bigger period is the period of materialization. If the periods overlap, then the period of materialization is a period that is formed by the union of the periods. A period of materialization is referred to as an official period. Definition 3.3.4 formally defines an official period.

Definition 3.3.4 (Official Period) *Given a history graph $H = (V, L, F)$, and a graph pattern $GP = (U, L, E, u_1, u_2)$, and two vertices in the history graph $v_1, v_2 \in V$, let S be discoverable(H, v_1, v_2, GP). We say that P is an official period when GP is matched against H at v_1 and v_2 if and only if:*

1. *there exists a subset S' of S, where $S' = \{P_1, \ldots, P_k\}$, so that $P = \bigcup S'$, and*
2. *there does not exists subset S'' of S and $P' \in \mathscr{P}$ such that (i) $S' \subset S''$, (ii) $P \subset P'$ and (iii) $P' = \bigcup S''$.*

official(H, u, v, G) is defined as the set of all official periods when graph pattern GP is matched against history graph H and u and v are vertices present in H.

Official periods are found using discoverable periods. If all discoverable periods are disjoint, meaning no two periods in the $\mathit{discoverable}(H, v_1, v_2, GP)$ overlaps with each other, then we call these periods official periods. In that case $\mathit{official}(H, u, v, G)$ is same as $\mathit{discoverable}(H, v_1, v_2, GP)$. However, when the discoverable periods have overlapping periods or one period containing another period, we take the overlapping periods' union to get an official period. We do this for all discoverable periods such that no two official periods overlap. In the case of a period containing another period, the union is the bigger period, and we do the same that we did for overlapping periods case to get official periods. Hence all periods in $\mathit{official}(H, u, v, G)$ are disjoint, and each representing a period for the materialization of the composite relationship.

Example 3.3.4 *In* Whatsapp Use Case*, we saw that when we try to find the composite relationship between a user and a message as described by the graph pattern and history graph in Figure 3.2, we get two discoverable periods* $[6, 10]$ *and* $[9, 15]$. *That means the composite relationship described by the graph pattern between* USER1 *and* MESSAGE1 *was active at periods* $[6, 10]$ *and* $[9, 15]$. *We can observe that these two periods overlap with each other at period* $[9, 10]$. *So the composite relationship was active between* $[6, 15]$*, which is the union of periods* $[6, 10]$ *and* $[9, 15]$. *The period* $[6, 15]$ *becomes the actual period for materialization and is referred to as an official period.*

In this section, we saw that how we can define a composite relationship using graph pattern. To find this composite relationship in the history graph, we need to solve a graph matching problem using a match. We get a match during some periods for the graph pattern against the history graph. We call discoverable periods and are the maximal periods during which the match was valid. Using these discoverable periods, we combine the periods such that we get distinct periods of materialization. These distinct periods of materialization of composite relationships are the official periods. In the next section, we will see the HO(T)-ReBAC policy language to define policies involving checking the temporal relations between the official periods where the temporal relations are defined using the 13 Allen relations.

3.4 HO(T)-ReBAC Policy Language

In the previous section, we discuss the periods of materialization for composite relationships. These composite relationships are used to define relationships between vertices in the history graph that are not related to each other directly. By comparing when these composite relationships materialized, we can define the access control policies. For example, let us define a user's presence in a room as a composite relationship and an IoT device presence in the same room by other composite relationships, then to have policies that compare when these composite relationships materialized, we would need to compare the official periods associated with these composite relationships. The policy may require user presence in the room to start before the presence of IoT devices in the room or vice versa. The policy may require the presence of a user in the room to overlap with the presence of an IoT device in the room. In HO(T)-ReBAC, we can create these policies by comparing the periods of materialization using temporal relations such as 13 Allen relations. This section will present the policy language in HO(T)-ReBAC for specifying how periods of materialization of composite relationships (i.e., official periods) are related to each other temporally. This policy language is called *HR*. In Section 3.4.1 we will describe the syntax of *HR* and Section 3.4.2 will describe the semantics of *HR*.

3.4.1 Policy Language Syntax

The policy language *HR* is the first-order logic. *HR* is designed for defining policies that can check if the found official periods are related to each other by Allen relations. The syntax of *HR* is used to determine whether a collection of symbols is a logical expression in *HR*. The symbols used in *HR* are variables, quantifiers, type declaration, atmoic formulae and logical connectives. The following provides details about each of the symbol in *HR*.

1. *Variables*: *HR* consists of two types of variables. The first type of variable is used to represent the vertices of graph patterns and is called *vertex variables*. We define *VVar* as the countable set for all vertex variables. Typical members of *VVar* are de-

noted by X, Y, Z, or their subscripted form. The second type of variables represents the official periods and are called *period variables*. *PVar* is defined as the countable set for all period variables. Typical members of *PVar* are denoted by I, J, K, or their subscripted form.

2. *Quantifiers*: *HR* has two quantifiers, (i) $\exists$ for existential quantification, and (ii) $\exists_\infty$ for special existential quantification. The existential quantification, $\exists$, is required because in HO(T)-ReBAC policies, we are interested in the existence of some official periods for the composite relationships in the policies such that they are related to each other through a given subset of Allen relations. The special existential quantifier $\exists_\infty$ is similar to $\exists$ but has one additional constraint. $\exists_\infty$ requires that the official periods for the composite relationships must be ongoing periods. Therefore, $\exists_\infty$ restricts the binding of the period variable to ongoing periods: that is, the periods that have not yet ended at the moment. HO(T)-ReBAC policies do not require all the official periods for the composite relationships to be related to each other through a given subset of Allen relations that is why we do not have universal quantifier $\forall$ in HO(T)-ReBAC policy language. Based on the quantification, we can have two types of variables (i) free variables and (ii) bound variables. Free variables are those that are not quantified, and bound variables are those that are quantified. In *HR* the vertex variables are free variables, and the period variables are bound variables.

3. *Type declaration*: The type $GP(X,Y)$ represent the graph pattern GP birooted at vertices corresponding to vertex variables X and Y. Graph pattern GP belongs to the universe of graph patterns (i.e., $Graph(U,L)$) where U is a countably infinite set of vertices, and L is the finite set of relation identifiers. The construct $I : GP(X,Y)$ declares that the period variable I will assume values that are official periods when the graph pattern $GP(X,Y)$ is matched against the history graph H.

4. *Atomic formulae*: In *HR*, the atomic formulae is used to compare two official peri-

ods using a non-empty subset of the 13 Allen relations. It can be written as I *op* J, where I and J are the period variables and *op* is a non-empty subset of the 13 Allen relations.

5. *Logical connectives*: *HR* has four logical connectives. $\wedge$ for conjunction, $\vee$ for disjunction, $\neg$ for negation, and $\top$ for always true.

The policy language consists mainly of two parts (i) *Prefix* and (ii) *Matrix*. The prefix part concerns the quantification of the official periods, and the matrix is concerned with checking that the quantified official periods are related to each other via some Allen relations. We describe *HR* language as $HR(VVar, PVar, U, L)$. The syntax for the policy, ϕ, is as follows:

$$\phi ::= Prefix\ .\ Matrix$$

$$Prefix ::= \varepsilon \mid \exists\, I : GP(X,Y)\ .\ Prefix \mid \exists_{\infty}\, I : GP(X,Y)\ .\ Prefix$$

$$Matrix ::= \top \mid I\ op\ J \mid \neg\, Matrix \mid Matrix_1 \wedge Matrix_2 \mid Matrix_1 \vee Matrix_2$$

3.4.2 Policy Language Semantics

The semantics of *HR* is defined in terms of the satisfaction relation $\models$. The satisfaction relation is defined in terms of two other relations, (i) $\Vdash$ and (ii) $\rhd$. The first relation $\Vdash$ checks if a formula is satisfied when a set of variable bindings are given. The second relation $\rhd$ construct the said variable binding environment.

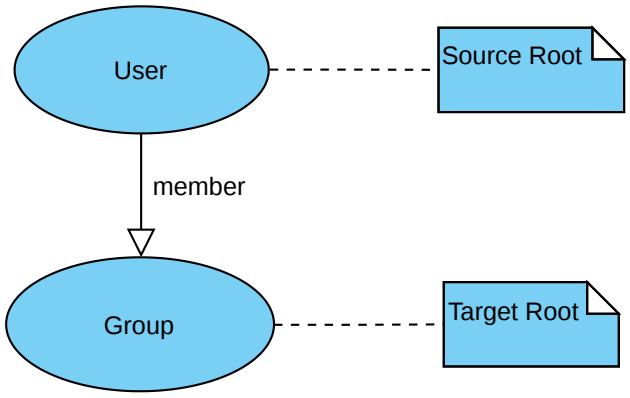
The relation $\Vdash$ checks if a matrix is satisfied under the assumption of some given variable bindings (for periods and vertex variables appearing in the matrix).

$$H, \Pi, env \Vdash Matrix$$

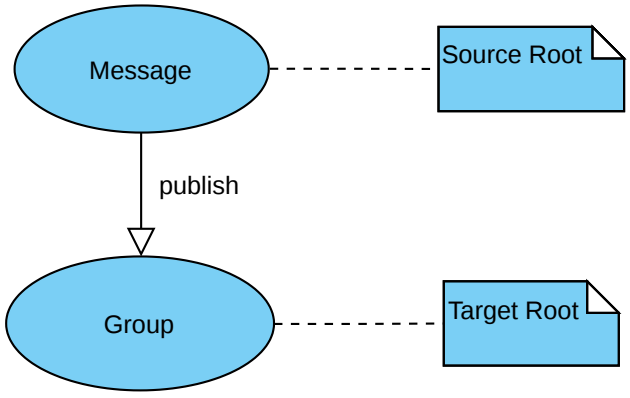
The sequent asserts that *Matrix* is satisfied under the context of H, Π, and *env*. Here H is the history graph, and Π and *env* binds values to the vertex and period variables appearing in *Matrix*.

Figure 3.3: Graph Patterns for Whatsapp Use Case Policy

(a) Graph Pattern GP_{UG}



(b) Graph Pattern GP_{MG}



The partial function $env : PVar \rightharpoonup \mathscr{P}$ maps period variables to actual periods. It is intended that $dom(env)$ is the set of period variables appearing in *Matrix*. Later on, when we discuss the $\triangleright$ relation, we will see how *env* is constructed in the first place.

Similarly, the partial function $\Pi : VVar \rightharpoonup V(H)$ maps vertex variables to vertices in the history graph. The intention is that $dom(\Pi)$ is the set of vertex variables appearing in *Matrix*. Unlike the period variable bindings, which are constructed via the relation $\triangleright$, bindings to the vertex variables are defined by the access request that is being authorized. For example, there will be a vertex variable identifying the requestor, another vertex variable identifying the resource. Such binding information is fixed by the access request itself.

The relation $H, \Pi, env \Vdash Matrix$ is defined as follows:

1. $H, \Pi, env \Vdash \top$ always holds.
2. $H, \Pi, env \Vdash Matrix_1 \wedge Matrix_2$ iff $H, \Pi, env \Vdash Matrix_1$ and $H, \Pi, env \Vdash Matrix_2$.
3. $H, \Pi, env \Vdash Matrix_1 \vee Matrix_2$ iff $H, \Pi, env \Vdash Matrix_1$ or $H, \Pi, env \Vdash Matrix_2$.
4. $H, \Pi, env \Vdash \neg Matrix$ iff it is not the case that $H, \Pi, env \Vdash Matrix$.
5. $H, \Pi, env \Vdash I\ op\ J$ iff there exists $r \in op$ such that $env(I)\ r\ env(J)$.

Points 1 to 4 in the above definition described when the relation $\Vdash$ holds for the various syntactic constructs: (i) True (ii) conjunction, (ii) disjunction, and (iv) negation. In point 5, the relation holds true if the official periods mapped to the period variables I and J are related to each other temporally through some Allen relation r where r is in *op* (which itself is a non-empty subset of the 13 Allen relations).

Relation $\triangleright$ construct the variable binding environment that binds the period variables to actual periods.

$$H, \Pi, env \triangleright Prefix$$

The sequent asserts that *env* is constructed using the context of H, Π, and *Prefix*. Here H is the history graph, and Π and *env* binds values to the vertex and period variables appearing in the

Prefix. We have detailed about Π and *env* in the discussion of relation $\Vdash$. To construct *env*, we find official periods for the graph pattern in *Prefix* and history graph H. Then we construct the partial function *env* by assigning one value at a time from the found official periods to the period variable.

The relation $H, \Pi, env \rhd Prefix$ is defined as follows:

1. $H, \Pi, \varnothing \rhd \varepsilon$ always holds.

2. $H, \Pi, env \rhd \exists\ I : GP(X,Y)\ .\ Prefix$ iff $H, \Pi, env' \rhd Prefix$ and $env = env'[I \mapsto P]$, where $P \in \mathit{official}(H, \Pi(X), \Pi(Y), GP)$.

3. $H, \Pi, env \rhd \exists_\infty\ I : GP(X,Y)\ .\ Prefix$ iff $H, \Pi, env' \rhd Prefix$ and $env = env'[I \mapsto P]$, where $P \in \mathit{official}(H, \Pi(X), \Pi(Y), GP)$ and $P \in P_\infty$.

Point 1 in above definition describe value of *Prefix* when $H, \Pi, env \rhd Prefix$ holds always. Intuitively this is when the policy does not require the existence of official periods. Point 2 shows the construction of *env* where the period variable has been assigned a value from the official periods found when the graph pattern was matched against the history graph. Similarly, Point 3 shows the construction of *env* where the period variable has been assigned a value that belongs to the official periods and also to the universe of ongoing periods (i.e., P_∞).

With $\Vdash$ and $\rhd$ defined, we are now ready to define the satisfaction relation $\models$ that gives our policy language its semantics. The relation $H, \Pi \models Prefix\ .\ Matrix$ is defined as follows:

$H, \Pi \models Prefix\ .\ Matrix$ holds iff there exists an environment $env : PVar \nrightarrow \mathscr{P}$ such that (a) $H, \Pi, env \rhd Prefix$, and (b) $H, \Pi, env \Vdash Matrix$

Typically, there is a family $V \in V(H)$ of resources that share the same access control policy. The policy answers the question of "Shall the access of $v \in V$ be permitted by a user $u \in V(H)$"? We encode such a policy by writing $\phi[X_1 \mapsto v_1, X_2 \mapsto v_2, \ldots X_k \mapsto v_k](Y,Z)$, where $\{X_1, \ldots, X_k, Y, Z\}$ is the set of all vertex variables appearing in ϕ, and the vertex variable Y represents the user who requests access, while the vertex variable Z represents the resource for which access is being

requested. When an access request by user u to resource v is presented to the reference monitor for authorization, the latter will look up the access control policy $\phi[X_1 \mapsto v_1, X_2 \mapsto v_2, \ldots X_k \mapsto v_k](Y,Z)$ relevant to the situation, and then check the sequent $H, \Pi \models \phi$, where H is the current history graph (i.e., the protection state), and $\Pi = \{X_1 \mapsto v_1, X_2 \mapsto v_2, \ldots, X_k \mapsto v_k, Y \mapsto u, Z \mapsto v\}$.

Example 3.4.1 *In* Whatsapp Use Case*, we can describe a policy in which a user access a message only if there exists some period when both the user and the message were part of the same group. The graph pattern GP_{UG} is used to describe a composite relation between a user and a group, and the graph pattern GP_{MG} describe a composite relation between a message and a group. The policy has an Allen relation subset op that intuitively describes an overlap between periods and op is equal to* {meets, overlaps, starts, during, finishes, met by, overlapped by, started by, contains, finished by, equality}. *The policy to describe user access to the message is given as below:*

***Whatsapp Access Policy**: $\exists\, I : GP_{UG}(X,Y)\ .\ \exists\, J : GP_{MG}(Z,Y)\ .\ I\ op\ J$*

This policy requires the existence of official periods for both graph patterns $GP_{UG}(X,Y)$ and $GP_{MG}(Z,Y)$ such that the any two official periods (one for GP_{UG} and the other for GP_{MG}) overlaps with each other. The two official periods overlap with they are related to each other through any one of the relations in the given subset of Allen relation (i.e., op = {meets, overlaps, starts, during, finishes, met by, overlapped by, started by, contains, finished by, equality}*).*

This chapter introduces HO(T)-ReBAC model. We first described the protection state (i.e., History graph) of the HO(T)-ReBAC model. We then explained how we could define composite relationships using graph patterns and find periods of materialization of these composite relationships using a match. After that, we presented the HO(T)-ReBAC policy language to define policies comparing the period of materialization using the 13 Allen relations. In the next chapter, we present two use cases where HO(T)-ReBAC model can be used. These use cases help establish the HO(T)-ReBAC model's usefulness in the real world.

Chapter 4

Use Case

4.1 Introduction

This chapter aims to explore real-world use cases of HO(T)-ReBAC. The two reasons for exploring real-world use cases of HO(T)-ReBAC are:

1. It helps showing the usefulness of Ho(T)-ReBAC.

2. Exploring how HO(T)-ReBAC can be used in the real world will help us best evaluate HO(T)-ReBAC model performance. In Chapter 6, this exploration will help us make decisions to evaluate the model empirically.

In Chapter 3, we talked about transient relationships and how HO(T)-ReBAC can be used in making authorization decisions in the presence of these transient relationships. In Section 4.2 of this chapter we will describe some of these transient relationships in the real world. This will help in giving some generic scenarios where HO(T)-ReBAC can be used.

In Section 4.3, we will explore a more concrete use case for HO(T)-ReBAC use. This use case is based on a scenario where staff, patients, and IoT devices in a medical setup interact transiently. That means the interactions between them are not permanent. We call this use case *Medical Internet of Things*.

Another exciting area to explore is making authorization decisions when groups are involved. We see people working in groups or teams all the time. Also, IoT devices can be deployed in groups, for example, a group of lights in a smart home. In the g-SIS work by [23], extensive work has already been done related to access control in group settings. g-SIS has defined several membership properties and membership renewal properties in involving certain operations related to group membership. In Section 4.4 we will see how HO(T)-ReBAC policies can be used to

emulate the membership properties and membership renewal properties in g-SIS. This emulation will help us understand the use of HO(T)-ReBAC in group settings.

To summarise, in this chapter, we will be looking at the following three different aspects to study the real-world application of HO(T)-ReBAC.

1. Transient relations in the real world (Section 4.2)
2. Medical Internet of Things use case (Section 4.3)
3. Emulation of g-SIS membership properties and membership renewal properties using HO(T)-ReBAC policies (Section 4.4)

In the following section, we will see the transient relationships that we can find in the real world.

4.2 Transient Relations in Real World

In ReBAC, when we consider relationships between users and devices, we do not consider the duration of those relationships. Whenever we need to make an access decision based on the current set of relations among different entities, but in the real world, these relationships do have a particular duration, and these relationships can be active for a certain period and then become inactive. After some time, it may become active again. By tracking all these transient relationships, we can have a more expressive access control model that bases authorization on current relationships and historical relationships.

As we have seen in Section 1.3 of Chapter 1, there are various temporal extensions of Role-Based Access Control (RBAC) [13, 21, 22]. In temporal RBAC, roles are activated during certain times and deactivated during other times. To authorize an access request, temporal RBAC checks if the required roles are activated when the access is being requested. Therefore, in temporal RBAC, the only focus point is to authorize or deny an access request at a specific time. There is no checking for the interactions (both current and past) between the entities in the system. However, there

are use cases when these current and historical interactions need to be considered while making an authorization decision. An example when historical interactions can be useful is "grant access to a device if the requester has co-located with the device in the past". In HO(T)-ReBAC, we can track all the interactions between the entities (both past and current) using the history graph. The authorization decisions in HO(T)-ReBAC are based on how entities are currently interacting with each other as well as how entities have interacted with each other in the past. HO(T)-ReBAC combines the aspects of both Relationship-Based and History-Bases Access Control which is beyond the scope of Temporal RBAC. We will see in Sections 4.3 and 4.4 how HO(T)-ReBAC policies can be employed to capture authorization conditions that are framed in terms of both current and historical interactions. In this section, we will explore some of the transient relationships found in the real world.

1. *Location*: A user being present at a location can be seen as a relationship between the user and the location. For example, a user entering a building starts a relationship with the building. Because of this relationship, the user is permitted to access resources like elevators while in the building. Subsequently, leaving can be seen as the termination of the relationship, and the user would no longer have access to the building's resources. This way, entering and leaving the building makes the relationship transient.

2. *Scheduling*: Scheduled events by nature are transient. An event would start at a fixed time and end once the duration of the schedule is over. For example, scheduling maintenance of devices present in a building, the person or group responsible for the maintenance has access to the devices during the maintenance schedule. This, in turn, means the maintainers' relationship with the devices is also transient.

3. *Subscription*: A subscription generally has a start and end time. Once a subscription has ended, users can renew their previous subscriptions. For example, in MS Office

365 [7], if an expired subscription is renewed within 30 days of expiry, the user gets back access to all the created during the expired subscription data.

4. *Physical State*: A particular physical state allows the user to access specific devices. A user in a particular physical state can be modeled as a relationship between a user and herself. Suppose a user may want to provide access to the location information of a fitness tracker like Fitbit [6], Apple Watch [3], only when they are doing physical activity like running or walking. This way, the access is given through a transient relationship between the user and herself. Since the physical state changes, so does the relationship.

This section provided real-world examples of transient relationships. Whenever we find these relationships, we can use HO(T)-ReBAC for authorization decisions. The next section will offer a more concrete example that uses HO(T)-ReBAC for authorization decisions. We will further discuss how we can use HO(T)-ReBAC policies to articulate conditions of access.

4.3 Medical Internet of Things

The two motivations for exploring HO(T)-ReBAC in a healthcare scenario are (i) the importance of securing medical records of patients and (ii) the presence of transient relationships in a healthcare environment. The next part will explore the various entities and transient relationships that can be found in a healthcare environment. This will be followed by exploring how HO(T)-ReBAC policies can be used in the same environment.

4.3.1 Medical IoT Entities and Transient Relationships

This section will explore the various entities and transient relationships that can be found in a healthcare environment. There are seven entities in a healthcare environment we are interested in for this use case. These entities are as follows:

1. *Clinicians*: These are the staff working in the hospital. The entities of this types are the actual doctors and nurses of the hospital. The entities of entity type *Clinician* are referred by the subscripted form of *Clinician* like $Clinician_1$, $Clinician_2$, $Clinician_3$,..., $Clinician_n$.

2. *Patients*: These are the individuals that are receiving treatment in the hospital. The entities of the entity type *Patient* are referred by the subscripted form of *Patient* like $Patient_1$, $Patient_2$, $Patient_3$,..., $Patient_n$.

3. *Roles*: These are the job profiles of different clinicians in the hospital. These roles can be of a doctor, nurse, administrator. The roles can be seen as groups of clinicians assigned to do a specific job. Each entity of type role can be viewed as a different group. In this example, we will consider *Doctor*, and *Nurse* as entities for the corresponding roles.

4. *Devices*: The medical devices used to treat and monitor patients are categorized as the instance of the entity type *Device*. Some of the examples of devices can be *Heart Rate Monitor*, *Ventilator*, *Blood Pressure Monitor*. The entities are represented by their subscripted form *Device* i.e., $Device_1$, $Device_2$,..., $Device_n$.

5. *Areas*: These are the physical locations inside the hospital. These physical locations in the real world could be *Emergency Rooms*, *ICUs*, *Operation Theatres*. The entities are represented by the subscripted form of *Area* i.e., $Area_1$, $Area_2$,..., $Area_n$.

6. *Shifts*: This refers to the continuous working shifts assigned to a clinician. In this example we consider *Morning*, *Evening*, *Night* for shift entities.

7. *Sensitivity Levels*: Every device has a sensitivity level for the information stored in it. We can consider one entity type for levels i.e., *Sensitivity* and the entities are represented by subscripted form of *Sensitivity* i.e., $Sensitivity_1$, $Sensitivity_2$,...,

Sensitivity$_n$. We consider higher the *Sensitivity* subscripted number, the higher the sensitive information is present at that level.

Next, we will present the transient relations between different entities that described the interactions between them. There are nine transient relations in a healthcare environment that we are interested in for this use case. These transient relations are as follows:

1. *belongs*: There are multiple job profiles that clinicians can have in a hospital. It can be a doctor, nurse, or administrator in our use case. Each clinician has a job role to perform, and the role that the clinician performs is depicted by the relation identifier *belongs* between a clinician and role. A clinician can have multiple roles by having a relation *belongs* with that role.

2. *designated*: Clinicians generally work in shifts, which generally are morning, evening, or night shifts. That means they have dedicated hours during the day to perform their duties. The relation identifier *designated* is used to identify the shift in which a clinician has been scheduled to perform duties.

3. *assigned*: Each device is designed to gather certain health information for a patient, and this information can be of different sensitivity. The relation identifier *assigned* is used to relate a device to the sensitivity level that the device captures the information for.

4. *allowed*: Every role corresponds to a sensitivity level, which means the clinicians belonging to the role can access the information of that sensitivity level. The relation identifier *allowed* is used to indicate the sensitivity level that a role corresponds to.

5. *scheduled*: Shifts can be restricted to specific physical locations in hospitals to limit access and better protect patients' information. We can represent this restriction

using *scheduled*, meaning the clinicians in a shift can only access areas that have the relation *scheduled* with that shift.

6. *deployed*: The physical presence of a device entity inside a physical location in a hospital (as denoted by an area entity) is represented through the relation identifier *deployed*.

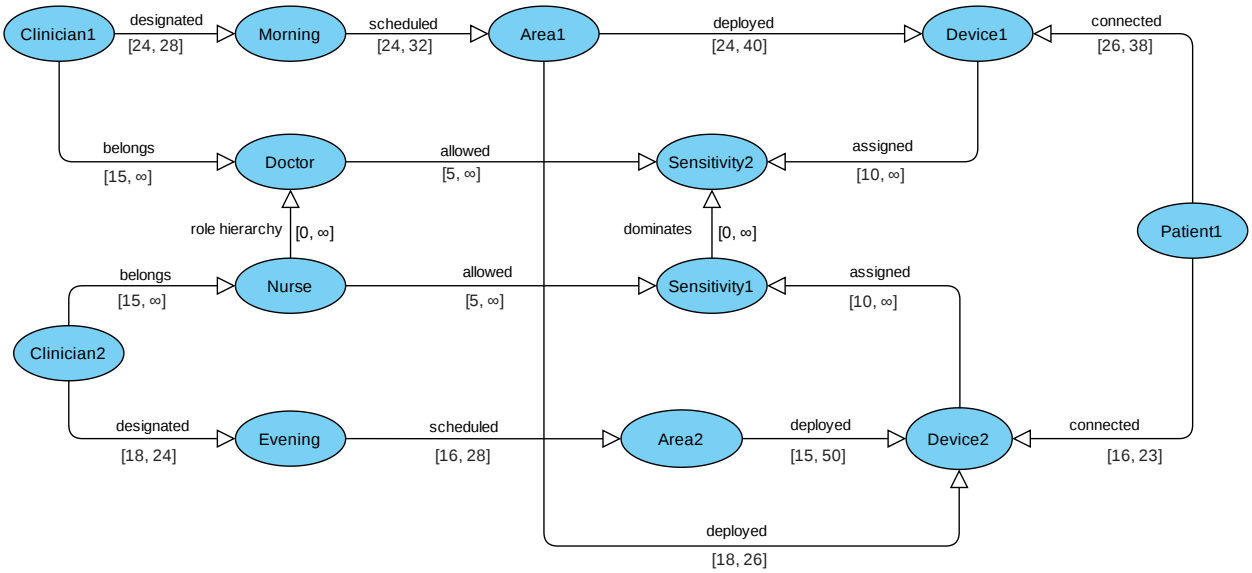
7. *connected*: If a device is attached to a patient to record health information, then the patient and device entities are related through the relation identifier *connected*.

8. *role hierarchy*: A role hierarchy relation identifier defines a hierarchy of access to resources between various *Roles*. A role hierarchy is transitive, meaning if $Role_1$ and $Role_2$ are related, and $Role_2$ and $Role_3$ are related, then $Role_1$ and $Role_2$ are related. It helps in cases where a role at a higher level may want to access resources accessed by a role at a lower-level role.

9. *dominates*: A dominates relation identifier defines hierarchies between various *Sensitivity Levels* that a requestor can access. Dominates is also transitive, meaning if $Sensitivity_1$ and $Sensitivity_2$ are related and $Sensitivity_2$ and $Sensitivity_3$ are related, then $Sensitivity_1$ and $Sensitivity_2$ are related. It helps in cases where a role wants to access a device at a lower sensitivity level if the role is allowed to access a device at a higher sensitivity level.

In a history graph, the entities are tracked as vertices, and the transient relationships are tracked as timed-relationships. Figure 4.1 shows different entities and the transient relationships in the healthcare environment tracked by a history graph. Some transient relationships are stable, and others are frequently changing. Both these stable and frequently changing relationships are expressed as timed-relationships in HO(T)-ReBAC. For example when a clinician is assigned a role, say a doctor role, it is not very frequent to change the clinician's role (e.g., $[15, \infty]$). In contrast, the presence of clinicians at a particular location in the hospital may be changed frequently (e.g.,

Figure 4.1: Medical IoT History Graph



$[24, 32]$). Both of these are expressed as timed-relationships in HO(T)-ReBAC. To emphasize, an ongoing period (e.g., $[15, \infty]$) does not mean permanence, but that the period's end has not yet been observed (thus ongoing).

In the next section using the entities and transient relationships discussed in this section, we will detail the use of HO(T)-ReBAC in a healthcare environment and discuss HO(T)-ReBAC policies for authorization decisions.

4.3.2 Medical IoT Use Case and Policies

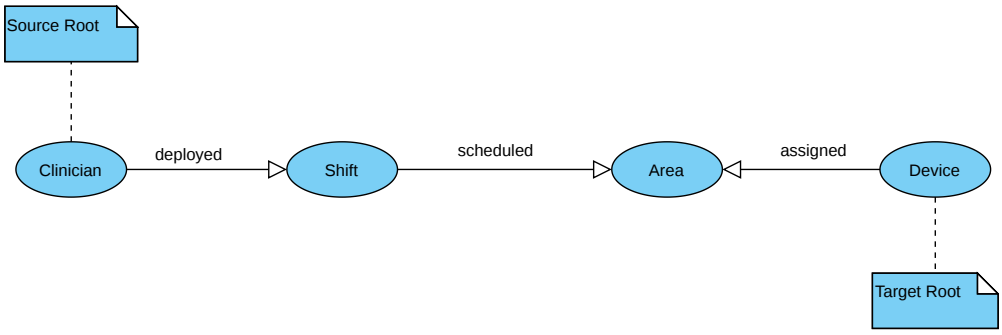
We are defining three graph patterns in the Medical IoT use case as shown in Figure 4.2. These three graph patterns define composite relationships between different entities in the Medical IoT use case. These three graph patterns are as follows:

1. Device Colocation: This graph pattern in Figure 4.2(a) is used for defining a composite relation between a clinician and a device that defines that the device and the clinicians are present in the same physical location.

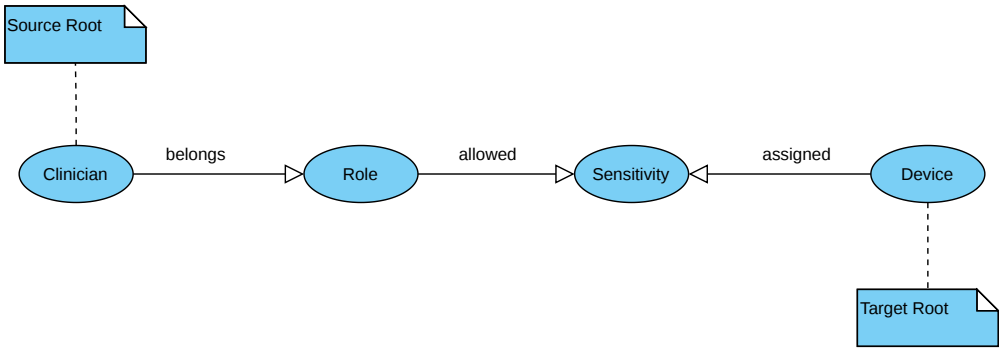
2. Device Authorization: This graph pattern in Figure 4.2(b) is used for defining a composite relation between a clinician and a device that defines that the clinician is

Figure 4.2: Medical IoT Graph Patterns

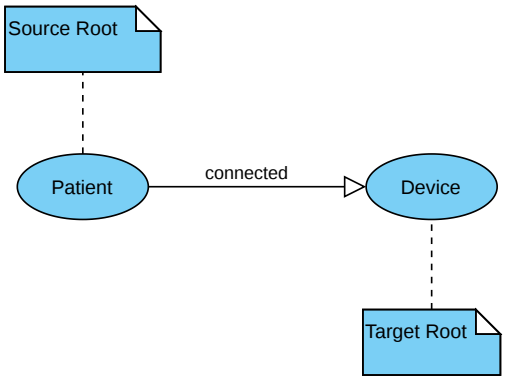
(a) Device Colocation Graph Pattern



(b) Device Authorization Graph Pattern



(c) Device Connection Graph Pattern



authorized to access the device.

3. Device Connection: This graph pattern in Figure 4.2(c) is used for defining a composite relation between a patient and a device that defines that the device is connected to the patient to gather her medical records.

In the HO(T)-ReBAC policies for Medical IoT, we use two Allen relations set op_{before} and $op_{overlapping}$. $I\ op_{before}\ J$ defines that the period I begins before the period J. The set op_{before} consists of Allen relations $\{$*precedes*, *meets*, *overlaps*, *contains*, *finished by*$\}$. Similarly, $I\ op_{overlapping}\ J$ defines that the period I overlaps with the period J. The set $op_{overlapping}$ consists of Allen relations $\{$*meets*, *overlaps*, *starts*, *during*, *finishes*, *met by*, *overlapped by*, *started by*, *contains*, *finished by*, *equality*$\}$.

Here we define three scenarios in Medical IoT and how we can use HO(T)-ReBAC for authorization in these scenarios:

1. **Real-time Device Monitoring**: Sometimes IoT devices are constraints with limited computation and storage. While using IoT devices, storing data can be a challenge, and it streams the data as soon as it is recorded. In *Real-time Device Monitoring* scenario, we will see how access can be granted to a clinician wanting to access these data streams. Here a clinician is trying to access data streamed by an IoT device monitoring a patient in real-time. This is useful when the monitoring devices have no way to store the patient's medical record data stream, and the clinician must be physically present near the device. Here are the four conditions that need to be valid for a clinician to access the data stream of the IoT device in real-time.

    (a) *Condition 1*: The clinician must be present in the same location as the device in order to access the data stream from the device.

    (b) *Condition 2*: For the clinician to be able to access the device, they must be authorized to access the device.

(c) *Condition 3*: For the device to record the patient's record, the device must be connected to the patient.

(d) *Condition 4*: All the three conditions should happen simultaneously, i.e., a period of overlapping must exist when the above three conditions were true. Also, all three conditions should happen in real-time.

In a HO(T)-ReBAC policy, we can check the above four conditions as follow. To check *Condition 1*, we use *Device Colocation* graph pattern (Figure 4.2 (a)). To check *Condition 2*, we use *Device Authorization* graph pattern (Figure 4.2 (b)). To check *Condition 3*, we use *Device Connection* graph pattern (Figure 4.2 (c)). To check *Condition 4*, we associate the ongoing existential quantifier i.e., $\exists_\infty$ with all three graph patterns (*Device Colocation*, *Device Authorization*, and *Device Connection*). This will ensure that we check for the existence of ongoing periods for all three graph patterns. Since all three graph patterns are associated with ongoing periods that would mean that in real-time the three periods are overlapping. Therefore we don not have to add condition for checking the overlapping for the three periods. Here is the HO(T)-ReBAC policy for **Real-time Device Monitoring**.

$$\begin{aligned} \textit{Policy 1} : \exists_\infty I : GP_{authorization}(Clinician, Device)\,. \\ \exists_\infty J : GP_{colocation}(Patient, Device)\,. \\ \exists_\infty K : GP_{connection}(Clinician, Device)\,.\,\top \end{aligned} \quad (4.1)$$

In *Policy 1*, the periods associated with period variables I, J, and K belong to the ongoing periods set. This signifies that the clinician that (i) the clinician and the device are present in the exact physical location, (ii) the clinician is authorized to gain access to the device, and (iii) the device is connected to the patient. All

these conditions are happening in real-time because the periods are ongoing periods. Since all the three conditions are happening in real-time, we can say that the three conditions overlap.

2. **Remote Access**: Using IoT devices gives an advantage of the device being connected to the Internet and thus can send data anywhere in the world. This can be useful in telemedicine where a clinician is not present in the exact physical location of the patient. *Remote Access* scenario is about a clinician wanting to access patient's data stored in an IoT device remotely. This is useful when a clinician cannot be present in the exact physical location of the device. Here the device can store the patient's record; therefore, it is not required for the device to be connected to the patient at the time of authorization. Three conditions need to be valid for a clinician to access a patient's data remotely.

    (a) *Condition 1*: For the clinician to be able to access the device, they must be authorized to access the device.

    (b) *Condition 2*: For the device to record the patient's record, the device must be connected to the patient.

    (c) *Condition 3*: The clinician should be authorized to access the device at the time of access, and there should exist an overlapping period when the device was connected to the patient, and the clinician was authorized to access the device.

To check *Condition 1*, we use *Device Authorization* graph pattern (Figure 4.2 (b)).To check *Condition 2*, we use *Device Connection* graph pattern (Figure 4.2 (c)). To check *Condition 3*, we associate the ongoing existential quantifier i.e., $\exists_\infty$ with *Device Authorization* graph pattern. In addition we have an atomic value i.e., $I\ op_{overlapping}\ J$ in the policy that checks for existence of an overlapping period for

the two graph patterns. Here is the HO(T)-ReBAC policy for **Remote Access**.

$$\textit{Policy 2} : \exists_{\infty}\, I : GP_{authorization}(Clinician, Device)\,.$$
$$\exists\, J : GP_{connection}(Patient, Device)\,.I\ op_{overlapping}\ J \quad (4.2)$$

In *Policy 2*, the periods associated with period variable I belong to the ongoing periods set. This signifies that the clinician is authorized to access the device at the time of authorization. $I\ op_{overlapping}\ J$ checks for the overlapping period when the device was connected to the patient, and the clinician was authorized to access the device.

3. **Medical Data Gathering**: In *Real-time Device Monitoring* scenario, we talked about accessing patients' data recorded by devices that are constraint by storage. There are IoT devices that have enough storage to keep patients' data. In *Medical Data Gathering* scenario, we will see how authorization decisions can be made scenario involving such devices. One example where we want devices to store the record is when a clinician goes on rounds for patients in the morning and wants to access the data recorded by the device for the previous day. In such cases, the clinician would like to go to the device connected to a patient and gather all the data recorded by the device before the clinician arrives. Here are the four conditions that need to be true for a clinician to gather data from an IoT device.

    (a) *Condition 1*: The clinician must be present in the same location as the device in order to gather data from the device.

    (b) *Condition 2*: For the clinician to be able to access the device, they must be authorized to access the device.

    (c) *Condition 3*: For the device to record the patient's record, the device must be connected to the patient.

(d) *Condition 4*: The clinician should be authorized to access the device at the time of access. Also, an overlapping period should exist when the device was connected to the patient and the clinician was authorized to access the device. Furthermore, the device should be connected to the patient to record data before the clinician arrives at the location to gather data.

To check *Condition 1*, we use *Device Colocation* graph pattern (Figure 4.2 (a)).To check *Condition 2*, we use *Device Authorization* graph pattern (Figure 4.2 (b)). *Condition 3*, we use *Device Connection* graph pattern (Figure 4.2 (c)). To check *Condition 4*, we associate the ongoing existential quantifier i.e., $\exists_\infty$ with *Device Authorization* graph pattern. In addition we have an atomic value i.e., $I\ op_{overlapping}\ J$ in the policy to ensure a period of overlapping when the clinician was authorized to access the device and the device was connected to the patient. We also have an atomic value $K\ op_{before}\ J$ to ensure that the device was connected to the patient before the clinician was present in the location. Here is the HO(T)-ReBAC policy for **Medical Data Gathering**.

$$\begin{aligned} \textit{Policy 3} : \exists_\infty I : GP_{authorization}(Clinician, Device)\,. \\ \exists\, J : GP_{colocation}(Clinician, Device)\,. \\ \exists\, K : GP_{connection}(Patient, Device)\,. \\ (I\ op_{overlapping}\ K) \wedge (K\ op_{before}\ J) \end{aligned} \quad (4.3)$$

In *Policy 3*, the periods associated with period variable I belong to the ongoing periods set. This signifies that the clinician is authorized to access to the device at the time of authorization. Periods associated with period variables J and K belongs indicates the device was connected to the patient, and the clinician was colocated

with the device. $I\ op_{overlapping}\ J$ ensures that the clinician was colocated with the device when they were authorized to access the device. $J\ op_{before}\ K$ ensures that the device was connected to the patient and the device gathered the data before the clinician accessed the device.

We have seen a concrete use case of HO(T)-ReBAC and saw how entities and transient relationships in a healthcare environment interact with each other. We also saw how HO(T)-ReBAC policies can be used for an authorization decision. Next, we will explore the use of HO(T)-ReBAC where groups are involved.

4.4 Group-Centric Secure Information Sharing Use Case

Group Centric Information Sharing (g-SIS) was developed by Krishnan et al. [23] to facilitate the sharing of information by bringing together users and objects in a group. g-SIS provides information on how authorization decisions can be made when users and objects are part of a group. In g-SIS, membership properties and membership renewal properties define the membership of users and objects in a group. This use case aims to see how we can utilize HO(T)-ReBAC in a group setting, i.e., users and objects are part of a group. We do this by emulating the membership and membership renewal properties of g-SIS in HO(T)-ReBAC. These properties are defined using 4 groups operations: Join, Add, Leave, and Remove. We will talk about these membership and membership renewal properties in the latter part of this section. In HO(T)-ReBAC, we use timed-relationships to emulate the g-SIS properties, and vertices in a history graph represent the users, objects, and groups.

In g-SIS, there are two variants (i.e., Strict and Liberal) for the 4 group operations in membership properties. Join and Leave operations are used for users, whereas Add and Remove operations are used for objects. Using Join and Add operations, respectively, a user and an object become part of a group. Using Leave and Remove, respectively, a user and an object cease to be part of a group. For each group operation, there are two variations, namely Strict and Liberal. We present

the difference between the Strict and Liberal variants of these operations in the following.

1. Strict Join (SJ) vs. Liberal Join (LJ): If a user joins a group using SJ, the user can only access objects added after the join time. However, if a user joins a group using LJ, the user may access objects added before the join time.

2. Strict Leave (SL) vs. Liberal Leave (LL): If a user leaves a group using SL, the user loses access to all objects. However, if a user leaves a group using LL, the user may retain access to some or all objects that the user was authorized to access before leave time.

3. Strict Add (SA) vs. Liberal Add (LA): If an object is added to a group using SA, the object may only be accessed by some or all of the users who joined before the Add time. However, if an object is added using LA, some or all users who joined after the Add time may also access the object.

4. Strict Remove (SR) vs. Liberal Remove (LR): If an object is removed from a group using SR, no user can access the object. However, if an object is removed from a group using LR, the object may still be accessed by some or all the users who had access to the object prior to the Leave time.

The Strict and Liberal variants of these operations are related to restrictiveness introduced for the access. Strict Join and Add variants are more restrictive and require a user to present an object in a group. Strict Leave and Remove variants remove all the subject's access over the object. Liberal Join and Add variants are less restrictive and do not require the subject to be present before an object in a group. Liberal Leave and Remove variants let the subject retain access over the object.

We have seen details about membership properties. Let us now explore more details about membership renewal properties. Membership renewal properties deal with access when a user

rejoins and subsequently leaves the group. The following are the 6 membership renewal properties in g-SIS.

1. *Lossless Join vs. Lossy Join*: In lossless join, a user does not lose any access held immediately prior to the join time, whereas in lossy join, a user loses some or all authorization held prior to the join.

2. *Nonrestorative Join vs. Restorative Join*: In nonrestorative join, past access may not be explicitly restored, whereas, in restorative join, past access is explicitly restored.

3. *Gainless Leave and Gainful Leave*: In gainless leave, access to new access can not be obtained by leaving a group, whereas, in gainful leave, new access may be given when a user leaves a group.

In the HO(T)-ReBAC policies for g-SIS membership properties and membership renewal properties, we use two Allen relations set $op_{overlapBefore}$ and $op_{overlapping}$. $I\ op_{overlapBefore}\ J$ defines that the period I begins before the period J and period I overlaps with period J. The set $op_{overlapBefore}$ consists of Allen relations $\{$*meets*, *overlaps*, *contains*, *finished by*$\}$. Similarly, $I\ op_{overlapping}\ J$ defines that the period I overlaps with the period J. The set $op_{overlapping}$ consists of Allen relations $\{$*meets*, *overlaps*, *starts*, *during*, *finishes*, *met by*, *overlapped by*, *started by*, *contains*, *finished by*, *equality*$\}$.

Here we have seen the membership properties and membership renewal properties in g-SIS and what they mean in terms of access. The following section will discuss how these properties can be emulated using the HO(T)-ReBAC policies.

4.4.1 g-SIS Membership Properties through HO(T)-ReBAC policies

This section will show how we can use HO(T)-ReBAC policies to emulate the actions of different membership properties and membership renewal properties. We will first describe the emulation of membership properties followed by the emulation of membership renewal properties.

Case No.	*Join*	*Add*	*Leave*	*Remove*	*HO(T)-ReBAC Policy*
1	Strict	Strict	Strict	Strict	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
2	Strict	Liberal	Strict	Strict	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
3	Liberal	Strict	Strict	Strict	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
4	Strict	Strict	Strict	Liberal	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
5	Strict	Liberal	Strict	Liberal	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
6	Liberal	Strict	Strict	Liberal	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
7	Strict	Strict	Liberal	Strict	$\exists I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
8	Strict	Liberal	Liberal	Strict	$\exists I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
9	Liberal	Strict	Liberal	Strict	$\exists I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
10	Strict	Strict	Liberal	Liberal	$\exists I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
11	Strict	Liberal	Liberal	Liberal	$\exists I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
12	Liberal	Strict	Liberal	Liberal	$\exists I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapBefore}\ J$
13	Liberal	Liberal	Strict	Strict	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapping}\ J$
14	Liberal	Liberal	Strict	Liberal	$\exists_\infty I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapping}\ J$
15	Liberal	Liberal	Liberal	Strict	$\exists I : GP_{member}(X,Y)\ .\ \exists_\infty J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapping}\ J$
16	Liberal	Liberal	Liberal	Liberal	$\exists I : GP_{member}(X,Y)\ .\ \exists J : GP_{publish}(Y,Z)\ .\ I\ op_{overlapping}\ J$

Table 4.1: *g-SIS* group membership properties cases

SNo.	*Join*	*Add*
1	Strict	Strict
2	Strict	Liberal
3	Liberal	Strict

Table 4.2: *g-SIS* join and add group operations with same meanings

Emulation Membership Properties: For emulating membership properties, we considered an instantiation of g-SIS. We picked a binary choice (Strict or Liberal) of the four group operations, i.e., Join, Add, Leave and Remove. The binary choices for the four operations give a total of 16 cases. We have enumerated these 16 cases in Table 4.1. We have also enumerated the HO(T)-ReBAC policies that enumerate these 16 cases in Table 4.1. The Strict/Liberal operations in Join/Add corresponds to the comparison between the membership periods in the group for the subject and the object. The Strict operation requires the subject to join a group before an object is added to the group. It also requires both periods of both these memberships to overlap. In comparison, the Strict/Liberal operations in Leave/Remove corresponds to whether the membership requires the subject or object to be present or not in the group at the time of authorization. This is checked using $\exists_\infty$ to check for ongoing periods. Strict operation requires the subject or object to be present in the group at authorization, whereas Liberal operation does not have any such requirements.

After studying these 16 cases, we found that we need 8 HO(T)-ReBAC policies instead of 16 HO(T)-ReBAC policies to emulate these 16 cases. We can reduce the number of HO(T)-ReBAC policies required because some cases have the same meaning and can be emulated using a single HO(T)-ReBAC policy. In Table 4.1, we have bundled the cases that have the same meaning. For example, we can see Case 1, Case 2, and Case 3 are bundled together, and there is a single HO(T)-ReBAC policy that emulates these 3 cases.

Some of these cases have the same meaning when at least one Join or Add operation is Strict. Strict operation is more restrictive than Liberal operation; therefore, when at least one of the Join/Add is Strict, the subject must join a group before the object was added to the group. 3 of the 4 combinations for Join and Add require the presence of subject before the object because of the presence of Strict operation in either Join or Add. Table 4.2 shows the 3 cases for Join and Add where the meaning is the same, i.e., presence of subject before the object in a group. As mentioned before, in Table 4.1, Case1, Case2, and Case3 correspond to the same HO(T)-ReBAC

policy. If we look at the pattern for Strict and Liberal for Join and Add for Case 1, Case 2, and Case 3, we can find that it matches with the Strict and Liberal pattern for Join and Add in Table 4.2. Similarly, cases 4, 5, and 6 mean the same, cases 7, 8, and 9 mean the same, and cases 10, 11, and 12 mean the same. Therefore, cases 1 to 12 correspond to 4 distinct meanings and can be represented by 4 distinct HO(T)-ReBAC policies. There is no Strict Join and Strict Add for Case 13, Case 14, Case 15, and Case 16. Therefore these all have distinct meanings. We have 4 distinct HO(T)-ReBAC policies for cases 13, 14, 15, and 16. Therefore, we have a total of 8 distinct HO(T)-ReBAC policies for the 16 g-SIS group operation cases.

Now we will see how we can emulate the membership renewal properties of g-SIS in HO(T)-ReBAC.

Emulation Membership Renewal Properties: From the membership renewal properties, we are only considering lossless join, restorative join, non-restorative join, and gainless leave. The reason for not considering lossy join and gainful leave is that in these two renewal properties, a user loses access by joining a group and gains access by leaving a group. This is non-monotonic and is not in accordance of group membership operations. Hence we will not consider these non-monotonic operations.

For the restorative, we can add an additional prefix in the policies described in Table 4.1 that requires a user to be present in the group at the time when access was requested. For example we can convert HO(T)-ReBAC policy for Case 12 in Table 4.1 to

$$Policy_{restorative} : \exists\, I : GP_{member}(X,Y)\,.\,\exists\, J : GP_{publish}(Y,Z)\,.$$
$$\exists_{\infty}\, K : GP_{member}(X,Y)\,.\, I\ op_{overlapBefore}\ J \quad (4.4)$$

Similarly, we can convert HO(T)-ReBAC policies for all remaining 15 cases to make them restorative. All properties in Table 4.1 are already lossless join, non-restorative join, gainless leave. They are lossless because, in all policies described in Table 4.1, a user does not lose any prior held access when they rejoin a group. The policies in Table 4.1 are non-restorative because, in these

policies, previous access is not explicitly restored. The policies in Table 4.1 are gainless because, in these policies, a user does not gain any access when it leaves a group. We have now seen how we can emulate the membership and membership renewal properties of g-SIS using HO(T)-ReBAC policies.

This chapter presents two use cases where HO(T)-ReBAC model can be used. The first use case detailed the usefulness of HO(T)-ReBAC in healthcare using Medical IoT use case. In the second use case, we showed how HO(T)-ReBAC policies can be used in group sharing scenarios by emulating g-SIS membership properties and membership renewal properties. The next chapter will see how we can find official periods in HO(T)-ReBAC to evaluate HO(T)-ReBAC policies.

Chapter 5

Finding Official Periods

5.1 Introduction

To evaluate HO(T)-ReBAC policies for authorization decisions, we must find official periods associated with matches from graph pattern to history graph. In this chapter, we will describe how we can find official periods and then how we get an authorization decision for a given HO(T)-ReBAC policy.

1. In Section 5.2, we will describe the problem of finding official periods as a Constraint Satisfaction Problem.

2. In Section 5.3, we will describe the algorithms that can efficiently find the official periods.

3. In Section 5.4, we will detail how the found official periods can be stored in database system so that we do not have to run the algorithms described in Section 5.3 to find the already found official periods.

4. In Section 5.5, we will detail how we translate a given HO(T)-ReBAC policy to a database query. The query is then executed against the stored official periods to get an authorization decision for the HO(T)-ReBAC policy.

Finding a match for the given graph patterns and history graphs amounts to a graph matching problem, which is a computationally hard problem to solve. The requirement in finding a match is that all timed-relationships that are mapped must overlap during some period, which adds additional constraints to the mapping problem. For HO(T)-ReBAC to be practical, the algorithm to find official periods needs to be as efficient as possible. In the past, the graph matching problem

has been solved by considering it as a Constraint Satisfaction Problem (CSP) [26]. We need first to formalize the problem of finding official periods as a CSP, which we will present in Section 5.2.

Once we have formalized our problem as a CSP, we need to modify existing CSP algorithms to incorporate overlapping period constraints. We will present the desired algorithm in Section 5.3. Once we have the algorithm, we will see how we can use this constraint to improve the algorithm's efficiency further.

Since solving a graph matching is a computationally hard problem, we should avoid finding the official periods that have already been found. Therefore we will introduce a mechanism to store the found official periods in Section 5.4. In Section 5.5 we will present an algorithm that translates a HO(T)-ReBAC policy to the query that can be executed on the database storing the official periods. We will also describe how executing the query can give the authorization decision for a HO(T)-ReBAC policy.

In the next section, we will formalize the problem of finding official periods as a CSP.

5.2 Finding Official Periods as Constraint Satisfaction Problem

Before we describe how finding official periods can be seen as a constraint satisfaction problem, we need to remember how official periods are found in HO(T)-ReBAC. For a given graph pattern and a history graph, to find official periods, we first need to find a Match for the given graph pattern and the history graph. A match is about finding a mapping (i) from the vertices in the graph pattern to the vertices in the history graph, and (ii) from the relationships in graph pattern to the timed-relationships in the history graph. Finding a match is a graph homomorphism problem with additional constraints. In graph homomorphism we ask, given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ does there exist a mapping $M : V_1 \to V_2$ such that $\forall u, v \in V_1$, if $(u, v) \in E_1$ then $(M(u), M(v)) \in E_2$? In finding a match, we need to find a mapping for the vertices and relationships in the graph pattern to the vertices and timed-relationships in the history graph. In addition to the mapping it is also required to find a nonempty period P such that P exists for all periods associated

with the mapped timed-relationships. We will refer to the problem of finding a match as *match existence*. To motivate describing finding official periods as Constraint Satisfaction Problem, we will first prove that *match existence* is an NP-Complete problem. Before starting the proof, we will define the graph homomorphism problem and the match existence problem.

Problem 1 ***Graph Homomorphism****:*

Instance *: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.*

Question *: Does there exist a mapping $M : V_1 \to V_2$ such that $\forall u, v \in V_1$, if $(u, v) \in E_1$ then $(M(u), M(v)) \in E_2$?*

Problem 2 ***Match Existence****:*

Instance*: A graph pattern $GP = (V_{GP}, L_{GP}, E_{GP}, u_1, u_2)$, a history graph $H = (V_H, L_H, E_H)$, 2 vertices $v_1, v_2 \in V_H$.*

Question*: Does there exist a nonempty period P and a match during P?*

Theorem 5.2.1 *The match existence problem is an NP-Complete problem.*

Proof: To prove the match existence problem is an NP-Complete problem, we have to prove (i) match existence problem is in NP, and (ii) match existence problem is an NP-hard problem.

Match existence problem is in NP*: Given a graph pattern $GP = (V_{GP}, L_{GP}, E_{GP}, u_1, u_2)$, a history graph $H = (V_H, L_H, E_H)$, 2 vertices $v_1, v_2 \in V_H$ and a match during nonempty period P, we can verify the match by checking (i) mapping for the vertices of graph pattern to the vertices of the history graph, (ii) mapping for the relationships of graph pattern to the timed-relationships of the history graph, and (iii) the nonempty period P exists for all periods associated with the mapped timed-relationships. We can check (i) in $O(|V_{GP}|)$, (ii) in $O(|E_{GP}|)$ time, and (iii) in $O(|E_{GP}|)$ time. Therefore we can verify the mapping M in $O(|V_{GP}| + (2 \times |E_{GP}|))$. Hence we can verify a solution to match existence problem in linear time. Hence the match existence problem is in NP.*

Match existence problem is an NP-hard problem*: To prove the match existence problem is an NP-hard, we will reduce an existing NP-hard problem to match existence problem. For this we will*

use graph homomorphism problem. The graph homomorphism problem is a known NP-Complete problem, therefore is NP-hard. Given a graph homomorphism problem instance $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$*, the reduction returns a graph pattern* $GP = (V_{GP}, L_{GP}, E_{GP}, u_1, u_2)$*, a history graph* $H = (V_H, L_H, E_H)$ *and 2 vertices* $v_1, v_2 \in V_H$*. The components of GP are defined as follows:*

1. $V_{GP} = V_1 \uplus \{u_1, u_2\}$ *where* $u_1, u_2 \notin V_1$ *are 2 distinguished vertices*
2. $L_{GP} = \{l, l_1, l_2\}$
3. $E_{GP} = (\bigcup_{(u,v)\in E_1} \{(u,v,l),(v,u,l)\}) \bigcup (\bigcup_{v\in V_1} \{(v,v,l_1)\}) \bigcup (\{(u_1,u_1,l_2),(u_2,u_2,l_2)\})$
4. u_1 *and* u_2 *are the 2 distinguished vertices mentioned above.*

The components of H are defined using a period $P = [0,1]$ *as follows:*

1. $V_H = V_2 \uplus \{v_1, v_2\}$ *where* $v_1, v_2 \notin V_2$ *are 2 distinguished vertices*
2. $L_H = L_{GP}$
3. $E_H = (\bigcup_{(u,v)\in E_2} \{(u,v,l,P),(v,u,l,P)\}) \bigcup (\bigcup_{v\in V_2} \{(v,v,l_1,P)\}) \bigcup (\{(v_1,v_1,l_2,P),(v_2,v_2,l_2,P)\})$

In the following, we will see how the generation of graph pattern GP described above is in linear time relative to the size of G_1*.*

1. *For each vertex in* G_1*, we generate a vertex in GP and also generate two distinguish vertices that are not present in* V_1*. Therefore we generate vertices of GP in* $O(|V_1| + 2)$ *time.*
2. *We then generate three relation identifiers* $\{l, l_1, l_2\}$ *for GP. This is done in* $O(3)$ *time.*
3. *For each edge in* G_1*, we generate two relationships in GP. Then for each vertex in* G_1 *we generate a relationship such that the generated relationship connects the vertex to itself. In addition, we create a relationship each for vertices* u_1 *and* u_2 *such that the generated relationship connects* u_1 *to itself as well as* u_2 *to itself. We generate the relationships in GP in* $O(|V_1| + (2 \times |E_1|) + 2)$*.*

4. *The time to generate a GP therefore is* $O((2\times|V_1|)+(2\times|E_1|)+7)$.

In the following, we will see how the generation of history graph H described above is in linear time relative to the size of G_2.

1. *For each vertex in* G_2*, we generate a vertex in H and also generate two distinguish vertices that are not present in* V_2*. Therefore we generate vertices of H in* $O(|V_2|+2)$ *time.*
2. *We relation identifiers for H are same as for GP. Therefore we don not have to generate relation identifier for H.*
3. *For each edge in* G_2*, we generate two timed-relationships in H. Every timed-relationship in H has the same associated period P which is* $[0,1]$*. Then for each vertex in* G_2 *we generate a timed-relationship such that the generated relationship connects the vertex to itself. In addition, we create a timed-relationship each for vertices* v_1 *and* v_2 *such that the generated timed-relationships connects* v_1 *to itself and* v_2 *to itself. We generate the timed-relationships in H in* $O(|V_2|+(2\times|E_2|)+2)$.
4. *The time to generate a GP is therefore* $O((2\times|V_1|)+(2\times|E_1|)+4)$.

Therefore we can reduce a graph homomorphism problem in polynomial time. Hence the match existence problem is an NP-hard problem. Since the match existence problem is in NP and the match existence problem is an NP-hard problem. Therefore the match existence problem is an NP-Complete problem. □

The match existence problem is a decision problem that requires a single mapping for a positive response. However, finding official periods requires searching for all maps that satisfy the match. Thus, finding official periods is at least as difficult as the match existence problem. Therefore we will formalize finding official periods problem as a CSP to use existing CSP algorithms or modify them to find official periods. As we have discussed in Section 2.5 of Chapter 2, a Constraint

Satisfaction Problem [31] consists of variables, domain for variables, and constraints among variables. We find official periods as CSP in which the vertices and relationships of a graph pattern are the variables. We refer the variables for the vertices as vertex variables and the variables for the relationships as relationships variables. The vertices in the history graph form the domain for the vertex variables, while the timed-relationships constitute the domain for the relationship variables.

There are three constraints between variables for finding official periods, namely (i) *Relation Identifier Constraint*, (ii) *Relationship-Vertex Constraint*, and (iii) *Period Constraint*. The following describes these constraints in details.

1. The *Relation Identifier Constraint* ensures that when a relationship in the graph pattern is mapped to a timed-relationship in the history graph, they have the same relation identifier.

2. The *Relationship-Vertex Constraint* is related to two ends for the relationship in the graph pattern and timed-relationships in the history graph. The images of the two ends of a relationship are the same as the two ends of the image of the relationship. If a relationship e is mapped to timed-relationship f, then the image of the two ends of e are the same the two ends of f.

3. For *Period Constraint*, when relationships in the graph pattern are mapped to the timed-relationships in the history graph, then *Period Constraint* requires the periods of all timed-relationships that belong to the codomain of the mapping to have a non-empty intersection.

Before formally defining the constraints, we would need to introduce some notations. Suppose we are given a graph pattern $GP = (U, L, E, u_1, u_2)$ where $u_1, u_2 \in U$, a history graph $H = (V, L, F)$, vertices $v_1, v_2 \in V$, $e \in E$, $f \in F$ and $l \in L$. $src(e)$ is the tail vertex of e in the graph pattern, $tgt(e)$ is the head vertex of e in the graph pattern, $l(e)$ is the relation identifier of e for the directed relationship e. $src(f)$ is the tail vertex of f in the history graph, $tgt(f)$ is the head vertex of f

in the history graph, $l(f)$ is the relation identifier of f, $P(f)$ is the period for the directed timed-relationship f.

Two map functions represent a solution to the CSP. These two map functions map (i) a vertex in the graph pattern to the vertex in the history graph, (ii) a relationship in the graph pattern to a timed-relationship in the history graph. $Map_{ver}(u)$ maps a vertex in the graph pattern to a vertex in the history graph and $Map_{rel}(e)$ maps a relationship in the graph pattern to a timed-relationship in the history graph.

Definition 5.2.1 (Relation Identifier Constraint (*RIC(e)*)) RIC(e) *is satisfied iff* $l(e) = l(Map_{rel}(e))$

Definition 5.2.2 (Relationship-Vertex Constraint (*RVC(e)*)) RVC(e) *is satisfied iff*

1. $Map_{ver}(src(e)) = src(Map_{rel}(e))$ *and*
2. $Map_{ver}(tgt(e)) = tgt(Map_{rel}(e))$

Definition 5.2.3 (Period Constraint (*PC(e_i, e_j)*)) *For* $e_i, e_j \in E$, PC(e_i, e_j) *is satisfied iff* $P(Map_{rel}(e_i)) \cap P(Map_{rel}(e_j)) \neq \varnothing$.

Lemma 5.2.2 (Special Case of Helly's Theorem) *Given a set* $S = \{P_1, P_2, P_3, \ldots, P_k\}$ *of periods such that all periods pairwise intersect with each other then there exists a common intersection of all k periods.*

Lemma 5.2.2 is a direct corollary to Helly's Theorem [19]. We can see that by keeping the number of dimensions in Helly's theorem equal to one, we get the result of pairwise intersecting convex sets having a common intersection. Since intervals are convex sets, we can say that a common intersection exists in a set of pairwise intersecting time intervals. Lemma 5.2.2 is significant because it allows us to use the *Period Constraint* defined in Definition 5.2.3, so we can say that whenever there is a set of timed-relationships such that every pair in the set has overlapping time periods, then there is a common intersection for all periods of timed-relationships in that set.

Theorem 5.2.3 *If* Map_{ver} *and* Map_{rel} *satisfy* RIC(e) *and* RVC(e) *for every* $e \in E$, *and satisfy* PC(e_i, e_j) *for every* $e_i, e_j \in E$, *then together* Map_{ver} *and* Map_{rel} *corresponds to a match.*

Figure 5.1: Constraint History Graph

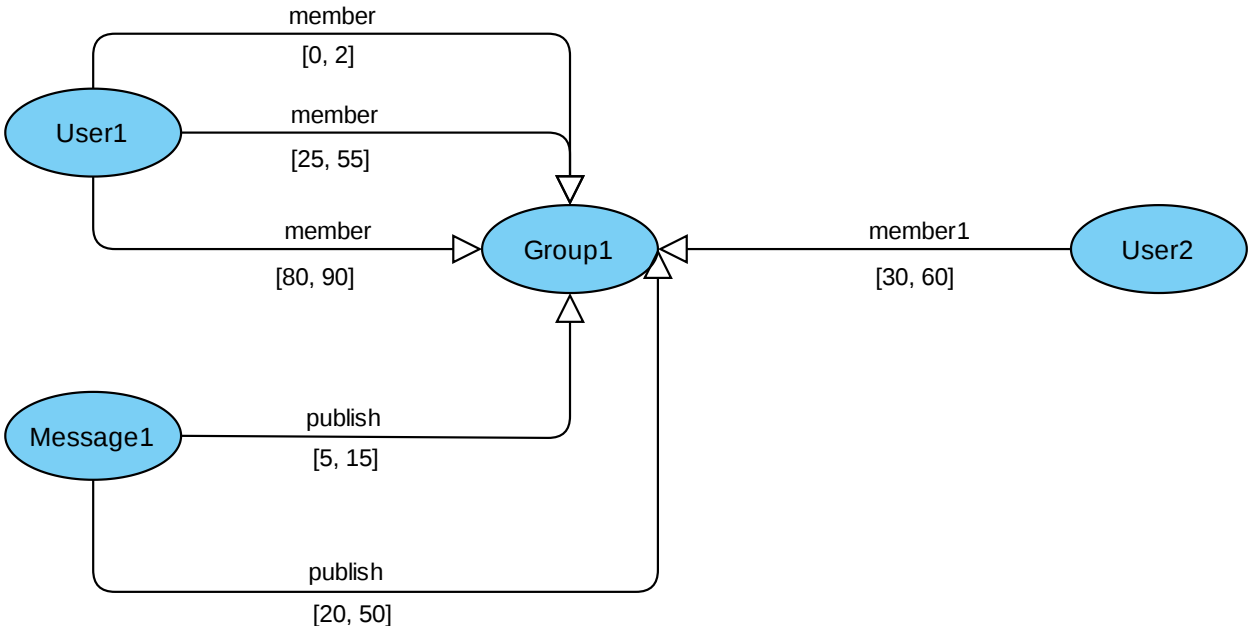
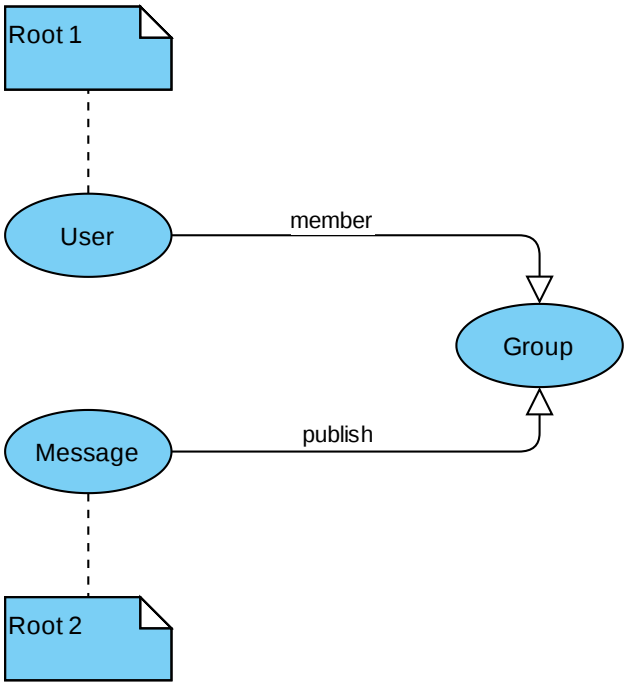


Figure 5.2: Constraint Graph Pattern



Proof: *For a graph pattern GP and a history graph H, a match during period P requires for every relationship in GP there exists a mapping to a timed-relationships in H such that an intersecting period P exists for all periods associated with the mapped timed-relationships.*

If Map_{ver} *and* Map_{rel} *satisfy* RIC(e) *and* RVC(e) *for every* $e \in E$*, then this ensures that the mapping for all vertices and relationships of GP to the vertices and timed-relationships of H exists.*

If Map_{ver} *and* Map_{rel} *satisfy* PC(e_i, e_j) *for every* $e_i, e_j \in E$*, then this ensures periods of the mapped timed-relationships has pairwise overlapping. Using Lemma 5.2.2, we say that is all periods of the mapped timed-relationships overlap with each other then there exists an intersecting period for all periods of the mapped timed-relationships. Hence satisfying the three constraints (*RIC(e)*,* RVC(e)*, and* PC(e_i, e_j)*) makes* Map_{ver} *and* Map_{rel} *a match.* □

The next part will further provide details about the three constraints using a chat group example.

Example 5.2.1 *In this example we will explore more about the three constraints using a history graph in Figure 5.1 and a graph pattern in Figure 5.1. The graph pattern includes vertices of type* User*,* Group *and* Message *and relationships with relation identifiers* member *and* publish*. The history graph includes vertices* User1 *and* User2 *of vertex type* User*,* Group1 *of vertex type* Group *and* Message1 *of vertex type* Message*. The history graph have the timed-relationships with relation identifiers* member*,* member1 *and* publish*. The following describes the four constraints using the given graph pattern and the history graph.*

1. Relation Identifier Constraint*: In the graph pattern there are two relationships i.e., (i)* member *between* User *and* Group *and (ii)* publish *between* Message *and* Group*. There are three timed-relationships between* User1 *and* Group1 *with relation identifier* member *for (i) and two timed-relationships between* Message1 *and* Group1 *with relation identifier* publish *for (ii) that satisfy the* Relation Identifier Constraint *because of the presence of same relation identifier between the relationships and the timed-relationships.*

2. Relationship-Vertex Constraint*: In the graph pattern for relationship* member *between* User *and* Group*, there are four timed-relationships, three between* User1 *and* Group1 *with relation identifier* member *that satisfy the* Relationship-Vertex Constraint*. For relationship* publish *between* Message *and* Group *there are two timed-relationships between* Message1 *and* Group1 *with relation identifier* publish *that satisfy the* Relationship-Vertex Constraint *because of the presence of same tail and head vertices and vertex type between the relationships and the timed-relationships.*

3. Period Constraint*: In the history graph there are three timed-relationships that pairwise overlap. These three time-relationships are (i)* member *between* User1 *and* Group1 *with period* $[25,55]$*, (ii)* publish *between* Message1 *and* Group1 *with period* $[20,50]$*, and (iii)* member1 *between* User2 *and* Group1 *with period* $[30,60]$*. The overlapping period for the three timed-relationships is* $[30,50]$.

In the next section, we will see how we can solve the problem of finding official periods using CSP algorithms.

5.3 Algorithm to Find Official Periods

In this section, we will describe the details to find official periods. Referring to Section 3.3.3 of Chapter 3, a period of materialization for a match is a discoverable period. For a given graph pattern and a history graph, we can find the set of discoverable periods by finding matches between the graph pattern and the history graph. We then convert this set of found discoverable periods to the set of official periods. We will provide the details of this process in three parts. The first part in Section 5.3.1 will describe Algorithm 9. Algorithm 9 converts a set of discoverable periods to a set of official periods. In the next part in Section 5.3.2, we will detail how we can find a match using existing CSP algorithms. In Section 5.3.4, we will optimise the solution in Section 5.3.2 so that we can find discoverable and official periods using a single algorithm.

5.3.1 Discoverable Periods to Official Periods

This section will describe an algorithm that converts a set of discoverable periods to a set of official periods. Section 3.3.4 of Chapter 3 defines official periods and describes in detail how an official period set is found using a set of discoverable periods. A discoverable period set is converted to an official period set when all the overlapping periods in the discoverable periods are consolidated until no periods overlap in the discoverable period set. We will use the algorithm in the following sections to find official periods after finding the discoverable periods through matches between a graph pattern and a history graph.

Algorithm 9: Discoverable Periods to Official Periods

```
   Input: Discoverable Period Set: DS ⊆ 2^P
   Local: P, P_0
 1 Initialize OS to ∅;
 2 Sort DS in ascending order of the start time of the periods;
 3 Let P_0 be the period in DS with the smallest start time;
 4 DS ← DS \ {P_0};
 5 foreach P ∈ DS in ascending start time do
 6 |   if P ∩ P_0 ≠ ∅ then
 7 |   |   P_0 ← P_0 ∪ P;
 8 |   else
 9 |   |   OS ← OS ∪ {P_0};
10 |   |   P_0 ← P;
11 OS ← OS ∪ {P_0};
12 return OS;
```

Algorithm 9 starts with two sets of periods (i) discoverable period set and (ii) official period set. The discoverable period set is the algorithm's input, and the official period set is the algorithm's output. The algorithm first sorts the discoverable period set in ascending order of the start time of the periods (line 1). The algorithm then removes the first period, P_0, i.e., the period with the smallest start time, from the discoverable set (line 3).

The algorithm then loops through the discoverable period set in the sorted order (lines 4-9). That means the first period that the algorithm loop through has the smallest start time after P_0. The algorithm then compares the period P_0 to the period P that has the next smallest start time

after P_0. The algorithm checks if there is an overlap between P_0 and P (line 5), then the algorithm consolidates the two periods into a single period which is the union of the two periods (line 6). Union of two periods means that the new consolidated period will have the smaller start time and the larger end time of the two periods. If there is no overlap between P_0 and P, the algorithm adds P_0 to the official period set (line 8). Then P becomes the period with the smallest start time and is assigned to P_0 (line 9). The algorithm then repeats this for the remaining periods in the discoverable period set. At the end of the loop, P_0 is either (i) a union of overlapping periods that includes the last period in the discoverable set or (ii) the last period in the discoverable set. The last period in the discoverable set is the period with the maximum start time (line 10). The algorithm adds P_0 to the official period set and then returns the official period set as the output of the algorithm (line 11).

To evaluate the algorithm's time complexity, let n be the number of periods in the discoverable period set. Sorting of discoverable periods takes $O(n \log n)$ time. Each operation in the loop for the discoverable period set is $O(1)$; therefore, the complexity for the loop for the discoverable period set is $O(n)$. So overall, this is a $O(n \log n)$ algorithm.

The following section will describe how we can find official periods using an existing CSP algorithm.

5.3.2 Official Periods using Existing CSP Algorithm

This section will detail finding official periods using a combination of an existing CSP Algorithm and Algorithm 9. In Section 5.2, we have formally described the problem of finding a match for a given graph pattern and a history graph as a CSP. Every match between the graph pattern and the history graph gives a discoverable period. To find all discoverable periods for a graph pattern and a history graph, we find all matches between the graph pattern and the history graph. We will then show the use of Algorithm 9 to convert the found discoverable periods to official periods.

There are existing algorithms to solve a CSP such as *Forward Checking (FC)* [25], *Forward Checking with Conflict-Directed Backjumping (FC-CBJ)* [25], and *Forward Checking augmented*

with Live-End Directed Backjumping (FC-LBJ) [26]. Rizvi and Fong [26] compared the performance of *FC*, *FC-CBJ*, and *FC-LBJ* for finding a mapping between two graphs with additional constraints. This is what we are trying to do in a match. Rizvi et al. found the *FC-LBJ*algorithm to be more efficient when compared to *FC* and *FC-CBJ*. Therefore, we will use the *FC-LBJ* algorithm for finding a match and thus discoverable periods. For finding official periods, we will combine *FC-LBJ* with Algorithm 9. We call this algorithm *FC-LBJ Baseline*.

We find official periods for the given graph pattern and the history graph as follows.

1. We first apply a CSP algorithm to solve the problem. In our case, we apply *FC-LBJ*.

2. The input for *FC-LBJ* are the variables, domain for the variables, and the constraint between the variables. *FC-LBJ* returns a set of the discoverable period as the output.

3. We provide the vertices and relationships in the graph pattern as the variables for the CSP. The vertices in the graph pattern are the vertex variables, and relationships in the graph pattern are the relationship variables.

4. We provide the vertices and timed-relationships in the history graph as the domain for the variables in CSP. The vertices in the history graph are the domain for a vertex variable, and the timed-relationships in the history graph are the domain for a relationship variable.

5. The three constraints described in the previous Section 5.2 namely (i) *Relation Identifier Constraint*, (ii) *Relationship-Vertex Constraint*, and (iii) *Period Constraint* are the three constraints between the variables in the CSP.

6. The variables, domain for the variables, and constraint between the variables are given as input to *FC-LBJ*.

7. When the *FC-LBJ* algorithm is run with the inputs, it finds all the matches between the graph pattern and the history graph. For each match, *FC-LBJ* calculates the pe-

riod of materialization, and as we know, the period of materialization for a match is a discoverable period. *FC-LBJ*, therefore, finds all discoverable periods and returns the discoverable period set as the output.

(a) We are not implementing a pure *FC-LBJ* in a strict forward checking manner. Constraint RVC is implemented in accordance of forward checking as a forward check. However, the period constraint is implemented as a backward check.

(b) Period is checked against the partial assignment for relationships. It is implemented by checking the overlapping with the periods of assigned variables by looking backward.

(c) We do it that way because the period constraint is a binary constraint between every pair of relationship variables. If we implement as forward check, then it will eliminate *CBJ*. To facilitate backjumping, we do it that way.

8. The found discoverable period set is given as the input to Algorithm 9. Algorithm 9 then returns the set of official periods as the output.

The *Period Constraint* requires the presence of a common overlapping period for all timed-relationships in the match. As we have discussed in Section 5.2, a common overlapping period is found when all the periods for the timed-relationships in the match pairwise overlap.

5.3.3 Official Periods using overlap tracker

We introduce a slightly improved implementation of *FC-LBJ Baseline* algorithm in which Period Constraint is enforced by a more efficient means. We refer this optimized algorithm as *FC-LBJ Overlap Tracking*.

1. We introduce a new global variable that tracks the common overlapping period for the partial solution of a match. This global variable will be referred to as *overlap tracker*.

2. When we try to map a relationship variable in the graph pattern to a timed-relationship in the history graph, then the period associated with the timed-relationship is checked against the *overlap tracker*.

3. If the period associated of the timed-relationship overlaps with *overlap tracker* then the timed-relationship is added to the partial solution of the match, and *overlap tracker* is updated. The *overlap tracker* is equal to the intersection of all the periods of the timed-relationships in the partial solution. As we construct a discoverable period, *overlap tracker* becomes smaller and smaller.

4. This optimization skips the step of *FC-LBJ Baseline* algorithm where we add a period to a set and then check for the pairwise overlapping for the entire set when we try to map a relationship variable in the graph pattern to a timed-relationship in the history graph.

5. As before, we use Algorithm 9 to compute the official period set from the discoverable period set.

We saw how *FC-LBJ Overlap Tracking* algorithm helps in reducing the calculations that we needed to do in *FC-LBJ Baseline* algorithm. *FC-LBJ Overlap Tracking* algorithm is more efficient when compared to *FC-LBJ Baseline* algorithm as in *FC-LBJ Baseline* algorithm when we need to assign a new value for a variables, we need to check for the overlapping with all previously assign values in the partial solution. So if there are k assign values in the partial solution we would need to check k time when assigning a new value. But in *FC-LBJ Overlap Tracking* algorithm we need to compare the new value with *overlap tracker* only so only 1 check is required. Therefore we reduced the number of check from k to 1 when assigning a new value for a variable. However, the

search tree in both algorithms remains same and therefore the number of partial assignments we end up exploring will be same.

5.3.4 Official Periods using Light Containment Check

This section will explore further improvements we introduce on top of *FC-LBJ Overlap Tracking* algorithm. Before we describe the improvements, recall what it means for a period to be contained in some other period. For two periods P_1 and P_2, P_1 is contained in P_2 means that P_1 is a subset of P_2. It means that the start time of P_1 is greater or equal to the start time of P_2 and the end time of P_1 is less than or equal to the end time of P_2.

In *FC-LBJ Overlap Tracking* algorithm, when we find a discoverable period we add it to an output discoverable period set. In some cases, the newly found discoverable period is contained in one or more periods present in the discoverable period set. In that case, the newly found discoverable period does not add any value in the finding of official periods. This is because with or without that period, the official period set would remain the same. In this improvement we will look for such discoverable periods that does not add value in finding official periods. Our aim is to recognise that a match will result in such a discoverable period as soon as possible and we can then skip finding that match. This involves checking for the discoverable periods contained in already found discoverable periods therefore we call this check as *Light Containment Check*. We refer to the algorithm as *FC-LBJ with Light Containment Check (*FC-LBJ-LCC*)*. Following provides details for *FC-LBJ-LCC*.

1. In *FC-LBJ-LCC*, we skip finding the match that will result in a discoverable period contained in any of the already found discoverable periods.

2. To check if the resultant discoverable period is already contained in already found discoverable periods, we compare the *overlap tracker* against the already found discoverable periods.

3. The *overlap tracker* tracks the partial solution for a match. If the partial solution is contained in an already found discoverable period, then the final discoverable period will also be contained in that discoverable period.

4. This way, we skip finding a match that would result in a discoverable period that does not add any value in finding official periods.

By introducing *Light Containment Check* in the *FC-LBJ-LCC* algorithm, we have reduced the search space for finding official periods by skipping redundant discoverable periods. We further improve the efficiency of *FC-LBJ-LCC* by directly comparing *overlap tracker* with the official period set instead of the discoverable period set.

5.3.5 Official Periods using Containment Check

The intuition behind this optimization is that some periods may not contained in any one of the discoverable periods in the discoverable period set but in the union of some overlapping discoverable periods in the discoverable period set. We know that the union of overlapping discoverable periods gives an official period. Therefore we compare *overlap tracker* with the official period set in order to skip finding periods that are contained in already found official period set. For doing so, we update the official period set as soon as we find a discoverable period using a modified version of Algorithm 9. We call this new algorithm *FC-LBJ with Containment Check (FC-LBJ-CC)*. The following provides details for the modification in Algorithm 9.

1. Modified Algorithm 9 takes the found discoverable period and a sorted set of official periods as input. Modified Algorithm 9 returns the updated sorted set of official periods as output. The official periods are sorted based on the start time of the periods.

2. Modified Algorithm 9 goes through sorted official periods and finds where the discoverable period can be added in the sorted official period set based on the start time of the discoverable period.

3. If the discoverable period does not overlap with any official period in the set, then we just add the discoverable to the official period set based on the start time of the discoverable period. Otherwise, if the discoverable period overlaps with an existing official period, we replace the existing official period by its union with the discoverable period. We call this set of overlapping periods $DS_{overlapped}$.

4. We repeatedly check for overlapping and merging the union of the two adjacent periods between $DS_{overlapped}$ and the next official period in the sorted official period set till no further overlapping exists or we have checked all official periods in the set. This ensures that all official periods remains disjoint.

5. Keeping the set of official periods in a set that is sorted all the time removes the necessity of sorting it repeatedly (see line 2 of Algorithm 9). This removes the sorting step and improves the complexity of the algorithm from $O(n \log n)$ to $O(n)$.

We started this section by describing how official periods can be found if given a set of discoverable periods. We then showed how to combine this with a CSP algorithm to find official periods for a graph pattern and a history graph. We improve this solution by introducing a tracker, namely *overlap tracker* and called this optimization *FC-LBJ Overlap Tracking* algorithm. We then introduce the containment check, which helps in skipping finding redundant discoverable periods. We called this optimize solution *FC-LBJ-LCC*. And then there is further improvement in *FC-LBJ-LCC* namely *FC-LBJ-CC*. In the next section, we will detail how we can store the found official periods and how these stored official periods can help evaluate HO(T)-ReBAC policies to make an authorization decision.

5.4 Storing Official Periods

In the previous section, we have detailed how we can find official periods by matching different graph patterns and against a history graph. A history graph is the protection state in HO(T)-ReBAC

Graph Pattern Identifier	*Official Period Start Time*	*Official Period End Time*

Table 5.1: Official Periods Table

Graph Pattern Identifier	T_{last}

Table 5.2: Last Checked Table

therefore all the official periods are found for one history graph and different graph patterns. In this match, we map the relationship of the graph pattern to the timed-relationships of the history graph and vertices of the graph pattern to vertices of the history graph. When the number of vertices and timed-relationships in the history graph increases, the domain for the vertex and relationship variables increases. As the domain size increases, it will take more time to find the match. Over a time, it will make HO(T)-ReBAC inefficient enough to be useful anymore. This motivated us to store the official periods that we have already found so that we do not have to recompute them repeatedly. To implement this, we first need to make changes required in *FC-LBJ-CC* so that the algorithm does not find the already found official periods. In the following steps, we detail the changes required in the *FC-LBJ-CC* algorithm.

1. We introduced a time point in the history graph representing the last time when official periods were found for the graph pattern. We call this time point T_{last}.

2. We provide T_{last} as an input to *FC-LBJ-CC* as input when finding official periods.

3. *FC-LBJ-CC* ensures that the domain for relationship variables consists only with the timed-relationships whose periods are active at or after T_{last}. Therefore *FC-LBJ-CC* uses T_{last} to ensure that all the official periods found are active at or after T_{last}.

After finding official periods using the updated *FC-LBJ-CC* algorithm, we store the found official periods for a graph pattern in a relational database. There are two tables in the database, namely *Official Periods Table* and *Last Checked Table*. *Official Periods Table*, stores the official

periods for the graph patterns. *Official Periods Table* as shown in Table 5.1 consists of three columns, one column to store the graph pattern identifier and other two to store the official period's start time and end time. *Last Checked Table*, stores the last time point when official periods were found for a graph pattern. *Last Checked Table* as shown in Table 5.2 consists of two columns, one to store the graph pattern identifier and the second to store the latest time point, i.e., T_{last} when official periods period for the graph pattern.

Algorithm 10: Update Database with Official Periods

Input: Graph Pattern, *GP* and History Graph, *H*

1 Retrieve T_{last} for the graph pattern from *Last Checked Table*;
2 **if** T_{last} *not found* **then**
3 | Initialize T_{last} to 0;
4 Run *FC-LBJ-CC* (H, GP, T_{last}) at T'_{last};
5 Add official periods for the graph patterns in *Official Periods Table*;
6 $T_{last} \leftarrow T'_{last}$;
7 Update T_{last} for graph pattern in *Last Checked Table*;

Algorithm 10 describes the steps to update the database with official periods that are found for a given graph pattern. Algorithm 10 takes as history graph, H and a graph pattern, GP as an input and updates the database with the official periods for H and GP. In the first step, we find the last time point for the graph pattern when official periods were found (line 1). If we do not find any such time point in the *Last Checked Table*, that means official periods are never found for the given graph pattern (line 2). In that case, we initialize the time point variable T_{last} to the initial point in the timeline that is 0 (line 3). We then find the official periods at the current time point, T'_{last}, using the *FC-LBJ-CC* algorithm by giving the history graph, H, graph pattern, GP, and the last time point, T_{last} as input (line 4). We add the found official periods in the *Official Periods Table* for the graph pattern, GP (line 5). At this time point, GP has all official periods that were active at or before T'_{last}. We updated the value of the last time point when official periods were found for GP to T'_{last} (line 6). Then we update *Last Checked Table* with the updated value of T_{last} for GP (line 7).

Using Algorithm 10, we can find official periods and cache them in a persistent data store

like a database. For a graph pattern *GP*, Algorithm 10 can run in a regular interval to cache the official periods for that interval. For example, Algorithm 10 can run at the end of each day for graph pattern *GP* and caches the official periods for *GP* that became active during that day. When we want to evaluate a policy that needs official periods for *GP*, we only the ones that became active since the last update of the cache for *GP*. Since Algorithm 10 runs every day for *GP*, the maximum of this interval will be equal to a day in this example. This helps in ensuring that finding official periods in HO(T)-ReBAC remains efficient even when the history graph size grows. In this section, we detailed how we store official periods after finding through *FC-LBJ-CC*. The next section will discuss how we can use these stored official periods to get an authorization decision for a HO(T)-ReBAC policy.

5.5 Authorization Decision in HO(T)-ReBAC

In the previous section, we have seen how we can cache the found official periods. This section will see how we can use these cached official periods to find authorization decisions for a HO(T)-ReBAC policy. There are two ways to find authorization decisions using the cached official periods. In the first method, we first retrieve all the official periods from the database for the graph patterns in the given policy and then compare these periods according to the Allen relations in the policy. In the second method, we convert the policy into an SQL query such that we only retrieve the official periods that follow the Allen relations constraint in the policy. We chose to go with the second method because it combines the retrieval and filtering of official periods in one step. The step involved in finding the authorization decision for a HO(T)-ReBAC policy are as follows:

1. Convert the HO(T)-ReBAC policy to an SQL query.
2. Execute the converted query on the database.
3. If the query returns at least one official period that satisfies all the policy constraints, then we grant authorization for the policy. Otherwise, we deny access.

The next part will detail how we can translate a given HO(T)-ReBAC policy to a database query.

5.5.1 HO(T)-ReBAC Policy to Database Query Translation

This part will describe the algorithm that translates a HO(T)-ReBAC policy to a database query that can be executed on the database to find an authorization decision. Since we are using a relational database management system (RDMS), we will be translating a HO(T)-ReBAC policy to an SQL query.

Algorithm 11: HO(T)-ReBAC Policy to SQL Translation

Input: HO(T)-ReBAC Policy: *Policy*

1 Convert *Policy* to Negation Normal Form;
2 Generate *SELECT* clause;
3 Generate *FROM* clause;
4 Generate *WHERE* clause;
5 SQLQuery ← Concatenate *SELECT* clause, *FROM* clause, and *WHERE* clause;
6 **return** *SQLQuery*;

Algorithm 11 is used to translate a HO(T)-ReBAC policy to an SQL query. Algorithm 11 gets the HO(T)-ReBAC policy as an input and returns the translated SQL query as output. The translated query consists of three clauses (i) *SELECT*, (ii) *FROM*, and (iii) *WHERE*. The *SELECT* clause corresponds to returning the found data from the database. We use it to return the results count found such that a 0 count means negative authorization since no result is found, and any count greater than 0 means a positive authorization, which corresponds to at least one result found. The *FROM* clause is used for translating prefix variables to SQL variables. Finally, the *WHERE* clause filters the database records based on the criteria specified by the policy.

The first step of Algorithm 11 is to convert the policy to Negation Normal Form (line 1). In Negation Normal Form, the negation symbol occurs before an atom in the formula. For a formula $\neg(A \wedge B)$ the equivalent Negation Normal Form will be $(\neg A \vee \neg B)$. A formula can be transformed in Negation Normal Form by pushing the $\neg$ inwards. We can transform a formula by applying the following equivalences from left to right.

$$\neg(\neg A) \equiv A$$

$$\neg(A \wedge B) \equiv ((\neg A \vee \neg B))$$

$$\neg(A \vee B) \equiv ((\neg A \wedge \neg B))$$

In HO(T)-ReBAC policy, the negation symbol occurs in front of the Allen relation set. A negation symbol in front of the Allen relation means taking the complement of the Allen relation set from the universal set of 13 Allen relations.

The next step to translate a HO(T)-ReBAC policy to an SQL query is to generate *SELECT*, *FROM*, and *WHERE* clauses (line 2-4) for the given HO(T)-ReBAC policy. The last step is to concatenate the *SELECT*, *FROM*, and *WHERE* clauses in the same order (line 5). The resultant statement from the concatenation gives the translated SQL query for a given HO(T)-ReBAC policy. In the following we will elaborate the generation of *SELECT*, *FROM*, and *WHERE* clauses.

Generate SELECT clause: The *SELECT* clause in the translated SQL query retrieves the number of records in the database that matches all the official periods' comparison checks. Therefore the *SELECT* clause is constructed by concatenating ***SELECT*** keyword with the number of results to be returned. This is written in SQL as

SELECT *COUNT(*)*

Generate FROM clause: To compare the two official periods with each other, we require two tables containing period values. We use a SQL *JOIN* clause that combines the rows from these two tables so we can compare the values. Since all periods are stores in *Official Periods Table*, we join *Official Periods Table* with itself. For each period variable we require Table 5.1 to have a *JOIN*. We can *JOIN* keyword to indicate a *JOIN* clause. To generate a *SELECT* clause, for each period variable x in the policy *prefix* we assign a SQL variable V_x. These variables are used as aliases for the *Official Periods Table* for each period variable x. We use the keyword ***AS*** to indicate as alias. We then perform a *JOIN* over all V_x using the keyword ***JOIN*** and keyword ***AS***. Finally, we concatenate ***FROM*** keyword followed with the created *JOIN* clause. For example for period variables x, y, z, we create SQL variables V_x, V_y, and V_z, then the generated *FROM* clause will be

Algorithm 12: MatrixTranslation

```
   Input: Matrix: M
   Global: MatrixString
1  switch M do
2      case T do
3          MatrixString = MatrixString + "TRUE"
4      case I op J do
5          MatrixString = MatrixString + PeriodComparison(I, op, J)
6      case M1 ∧ M2 do
7          MatrixString = MatrixString + MatrixTranslation(M1) + "AND" +
             MatrixTranslation(M2)
8      case M1 ∨ M2 do
9          MatrixString = MatrixString + MatrixTranslation(M1) + "OR" +
             MatrixTranslation(M2)
10 return MatrixString;
```

FROM *Official Periods Table* ***AS*** V_x ***JOIN*** *Official Periods Table* ***AS*** V_y ***JOIN*** *Official Periods Table* ***AS*** V_z

Generate WHERE clause: The *WHERE* clause consists of conditions to filter the SQL query results. These conditions come from checking for (i) ongoing periods and (ii) Allen relations in official periods. For the ongoing periods, the condition is to check for the period variables to be equal to the proxy value for ∞, which is the maximum value of an integer in SQL. To check for the Allen relations in official periods, we need to translate the policy *matrix* to SQL clause. Algorithm 12 is a recursive algorithm describing the translation from a policy *matrix* to an SQL clause. Algorithm 12 takes a policy *matrix* as an input and returns the translated SQL clause for the *matrix*. We maintain a global string variable *MatrixString*. The string is recursively concatenated with the SQL conditions we get from the *matrix*. We use $+$ symbol to concatenate two strings. Recall that the policy is first converted to Negation Normal Form. Therefore we only need to elaborate the following four matrix constructors (i.e., $\top$, *I op J*, $\wedge$, $\vee$).

1. $\top$: In this case, we add SQL condition true using the ***TRUE*** keyword indicating this condition always holds.

2. *I op J*: *I op J* holds if *I* and *J* are related to each other through one of the Allen

Relation in the Allen Relation set op. Function *PeriodComparison* adds SQL conditions to compare periods for periods variables I and J for the Allen relations in op. To check whether two periods are related to each other through some Allen relation, we compare their start and end times which are stored in Table 5.1. Since we only want one relation in op to satisfy between I and J, we use the keyword ***OR*** between all comparisons for Allen relations in op.

3. $M_1 \wedge M_2$: For this we recursively find conditions clause for matrix M_1. We concatenate the condition clause for M_1 with the SQL keyword ***AND*** and the resultant string is concatenated with condition clause we recursively find for matrix M_2.

4. $M_1 \wedge M_2$: For this we recursively find conditions clause for matrix M_1. We concatenate the condition clause for M_1 with the SQL keyword ***OR*** and the resultant string is concatenated with condition clause we recursively find for matrix M_2.

We have seen the way can generate *SELECT*, *FROM*, and *WHERE* clauses (lines 2-4) in Algorithm 11. Algorithm 11 translate a HO(T)-ReBAC policy to an SQL query which is executed against a database containing the official periods. The number of records returned in the query execution will determine whether the access will be granted or denied.

This chapter first explored how finding official periods can be described as a Constraint Satisfaction Problem. We then explored the methods to find official periods. We started with a solution using an existing CSP algorithm and then improved to skip finding redundant solutions. After that, we describe the methods for storing the official periods such that we do not have to find the official periods found in the past. This was followed by an algorithm to translate a HO(T)-ReBAC policy to an SQL query. This SQL query is then executed against the database storing official periods to find the authorization decisions. In the next chapter, we will empirically evaluate the performance of (i) algorithm to find official periods and (ii) policy evaluation.

Chapter 6

Empirical Evaluation

6.1 Introduction

In this chapter, we evaluate the performance of the HO(T)-ReBAC model's implementation. Two sets of experiments were conducted for the performance evaluation. The first set of experiments were conducted to measure the performance of *FC-LBJ with Containment Check (FC-LBJ-CC)* algorithm in finding the official periods. We call this set of experiments *FC-LBJ-CC evaluation*. For *FC-LBJ-CC evaluation*, we need to generate synthetic data such as history graphs and graph patterns. Section 6.2 will provide details for the experimental setup followed by the evaluation results for *FC-LBJ-CC evaluation*. After finding official periods, we conducted a second set of experiments to measure the performance of policy evaluation in HO(T)-ReBAC. We use the found official periods to compute authorization decisions for the HO(T)-ReBAC policies. We call this set of experiments *Policy evaluation*. For *FC-LBJ-CC evaluation*, we need to generate HO(T)-ReBAC policies and history graphs. Section 6.3 will provide details for the experimental setup followed by the evaluation results for *Policy evaluation*. In *Policy evaluation*, we will study the time it takes to get an authorization decision for a policy when it is evaluated using the stored official periods. *FC-LBJ-CC evaluation* and *Policy evaluation* together help to evaluate the performance of HO(T)-ReBAC model. Therefore, HO(T)-ReBAC performance evaluation involves finding the official periods to store in a data store and making authorization decisions for the given policies using the stored official periods. The performance evaluation will help determine whether HO(T)-ReBAC is practical.

We now turn to *FC-LBJ-CC evaluation* with experimental setup and empirical results.

6.2 FC-LBJ-CC Evaluation

In this section we will evaluate the performance of *FC-LBJ-CC* algorithm (Section 5.3.5). There are two motivation for *FC-LBJ-CC evaluation*. The first is to measure the performance of *FC-LBJ-CC* algorithm and the second is to compare *FC-LBJ-CC* algorithm (Section 5.3.5) with *FC-LBJ Baseline* algorithm (Section 5.3.2) to find official periods. This section consists of the experimental setup for the evaluation (Section 6.2.1) followed by the empirical results and discussion for *FC-LBJ-CC evaluation* (Section 6.2.2).

6.2.1 FC-LBJ-CC Evaluation Experimental Setup

In this section, we will first describe the configuration of the system we used for *FC-LBJ-CC evaluation*. This will be followed by the details of history graphs and graph patterns we generated for conducting the experiments.

System Configuration: *FC-LBJ-CC evaluation* was carried on an Amazon Web Services EC2 T2.2Xlarge instance [2] running Ubuntu operating system with 32 GB RAM, 80 GB of SSD storage, and Intel Xeon 8-core processors having 2.3 GHz clock speed. Since the history graph is a graph, we chose Neo4j version 3.4.0 [9], a graph database to store the history graph. The algorithm *FC-LBJ-CC* and graph patterns were implemented on Java 11.

To evaluate *FC-LBJ-CC* we require history graphs and graph patterns. We use social graphs as the basis for generating the history graphs. We used the created history graphs to generate graph patterns so that at least one official period exists when the graph pattern is matched against that history graph.

History Graph Generation: To create a history graph, we used a social network as the base graph. The social network we are using is Epinions social network [27] from Stanford Large Network Dataset Collection [10]. The directed social graph consists of 75,879 nodes and 508,837 edges. The vertices of the social graph are one-to-one mapped to the vertices of our history graph. Following Rizvi and Fong [28, 29], this history graph has 7 relation identifiers. The edges are

converted to the timed-relationships using two variables (i) **Total timeline interval,** T, and (ii) **Probability of event actualization,** P_a.

Here are the steps involved to create history graphs using the two variables T and P_a.

1. There are $T+1$ time points with $T-1$ unit periods like $[0,1],[1,2],[2,3].....[T-2,T-1]$ and a special period $[T-1,\infty]$ to represent an ongoing period. Therefore we have a total of T periods ($T-1$ unit periods and one ongoing period).

2. For each relationship edge e and for each of the T periods P, we do the following.

    (a) We randomly chose a relation identifier l from the 7 relation identifiers.

    (b) With probability P_a, the relationship e is designated active during the periods P. That is with probability $(1-P_a)$ the relationship e is designated inactive during period P.

    (c) Consecutive periods are consolidated into a continuous period.

    (d) For each of the periods formed after consolidation, we assign a timed-relationship for the edge with that consolidated period and the relation identifier l.

Now that we have seen how we can use variables T and P_a, we will describe the values of T and P_a we have used to generate the history graphs. We envision the system using HO(T)-ReBAC will run once every day to cache the official periods, so we divided the discrete timeline in the number of hours per day; therefore, the value of T we used is 24. By keeping the value of T constant and varying values of P_a from 0 to 1, we created multiple history graphs with varying interactions between the vertices in the history graph.

Graph Pattern Generation: For every history graph created, we generated random graph patterns such that there exists a match in the history graph for every generated graph pattern. This

is to ensure that we find official periods for the graph pattern for which we are evaluating. We created graph patterns for the experiment as follows:

1. We take four input parameters to generate a graph pattern: (i) N_{ver}, number of vertices in the graph pattern, (ii) a history graph, (iii) $T_{timeout}$, timeout limit, and (iv) *periodSet*, a set of periods, containing T periods of which $T-1$ are unit periods, and one is an ongoing period. The algorithm returns the generated graph pattern as output.

2. We maintain two sets of vertices, *verSet*, and *tempVerSet* and a set of timed-relationships, *timedRelSet*, that are used to generate a graph pattern.

3. We first chose a period P at random from the *periodSet*. In the history graph, we keep only the timed-relationships that overlapped with P and remove all other timed-relationships. We call this *history graph snapshot*.

4. We then select a vertex at random from the history graph snapshot and add it to the *verSet*. We randomly choose a vertex from *verSet* and find all its neighbours in the history graph snapshot. We add the found neighbours to *tempVerSet*. We randomly select a vertex from *tempVerSet* and add it to *verSet* if it is not already present. We repeat this till the size of *verSet* equals N_{ver}. If we can not find the required number vertices in *verSet* within $T_{timeout}$, we go back to Step 3.

5. We then add the timed-relationships between the vertices in *verSet* to *timedRelSet*. We generate vertices of the graph pattern from the corresponding vertices in *verSet* and relationships of the graph pattern from the corresponding timed-relationships in *timedRelSet*. We then randomly choose the two vertices from the graph pattern and make them the graph pattern's roots.

Now that we have seen the setup for the *FC-LBJ-CC evaluation*, in the next section, we will

examine the experimental results for *FC-LBJ-CC evaluation* and discuss the significance of these results.

P_a	*FC-LBJ-CC*	*FC-LBJ Baseline*
0.1	1000	1000
0.2	1000	999
0.3	1000	995
0.4	999	994
0.5	997	992
0.6	1000	995
0.7	1000	996
0.8	1000	998
0.9	1000	998
1.0	1000	1000

Table 6.1: *FC-LBJ-CC* and *FC-LBJ Baseline* Completed Runs

6.2.2 FC-LBJ-CC Evaluation Measurements and Results

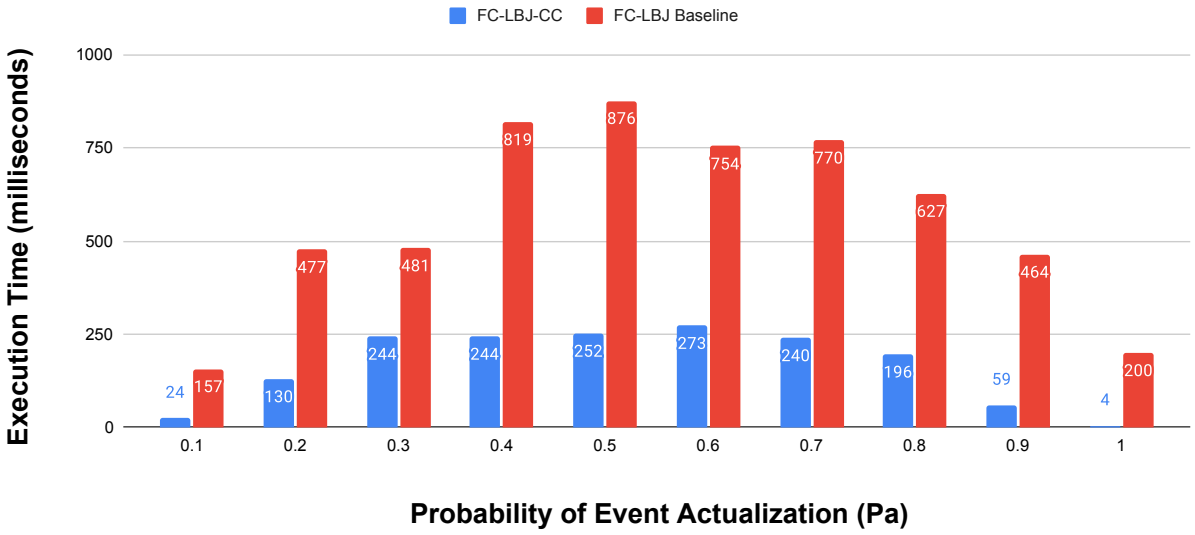
FC-LBJ-CC Evaluation Measurements: We have created history graphs and graph patterns for *FC-LBJ-CC Evaluation*. Using *FC-LBJ-CC* we find the official periods, and our main concern was the time it takes for *FC-LBJ-CC* to find the official periods. We were also interested to compare *FC-LBJ-CC* and *FC-LBJ Baseline* algorithms to study the effect of introducing containment check in *FC-LBJ Baseline*. We measure these effects by varying the interactions between vertices in a history graph.

Time taken for finding official periods will help determine whether the model is efficient enough for real-world applications. For each value of P_a, we generated a new history graph. Therefore we created 10 different history graphs by increasing the value of P_a from 0.1 to 1 with an increment of 0.1. Then we needed to generate graph patterns and for that we first needed to determine the number of vertices in the graph pattern. For this, we referred back to our Medical IoT use case in Chapter 4. We found the maximum vertices in the graph patterns to be 4. However, in our experiments, we increased this to 6 vertices in graph patterns to further stress-test *FC-LBJ-CC Evaluation*. For each history graph, we generated 1,000 random graph patterns. We measured the

execution time for both *FC-LBJ-CC* and *FC-LBJ Baseline* for the generated 1,000 graph patterns. Therefore the total data points we measure were 10 $\times$ 1,000 = 10,000 to compare *FC-LBJ-CC* and *FC-LBJ Baseline*.

We have used a timeout of 60,000 milliseconds for the measurements. We stopped that run and moved on to the next execution for an execution that exceeded the timeout. Table 6.1 compares the number of runs completed within the given timeout in *FC-LBJ-CC* and *FC-LBJ Baseline* algorithms. Using Table 6.1 we observe improvements in finding the results within the timeout in *FC-LBJ-CC* when compared to *FC-LBJ Baseline*. Out of the total 10,000 runs, *FC-LBJ-CC* was not able to find results in 4 executions within the timeout; this is an improvement over *FC-LBJ Baseline* where the number execution not able to find results within the timeout is 33. This is a first indication that *FC-LBJ-CC* is more efficient in finding official periods when compared with *FC-LBJ Baseline*. We will explore more about the execution time of both algorithms in the *FC-LBJ-CC Evaluation Results* part. When measuring the average time of execution for *FC-LBJ-CC* and *FC-LBJ Baseline* algorithms, we only used executions that were completed within the timeout.

Figure 6.1: *FC-LBJ-CC* and *FC-LBJ Baseline* Comparison



FC-LBJ-CC Evaluation Results: Figure 6.1 show the result for execution time to find results for both *FC-LBJ-CC* and *FC-LBJ Baseline*. We can observe the change in execution time for

Figure 6.2: Number of Official Periods Found

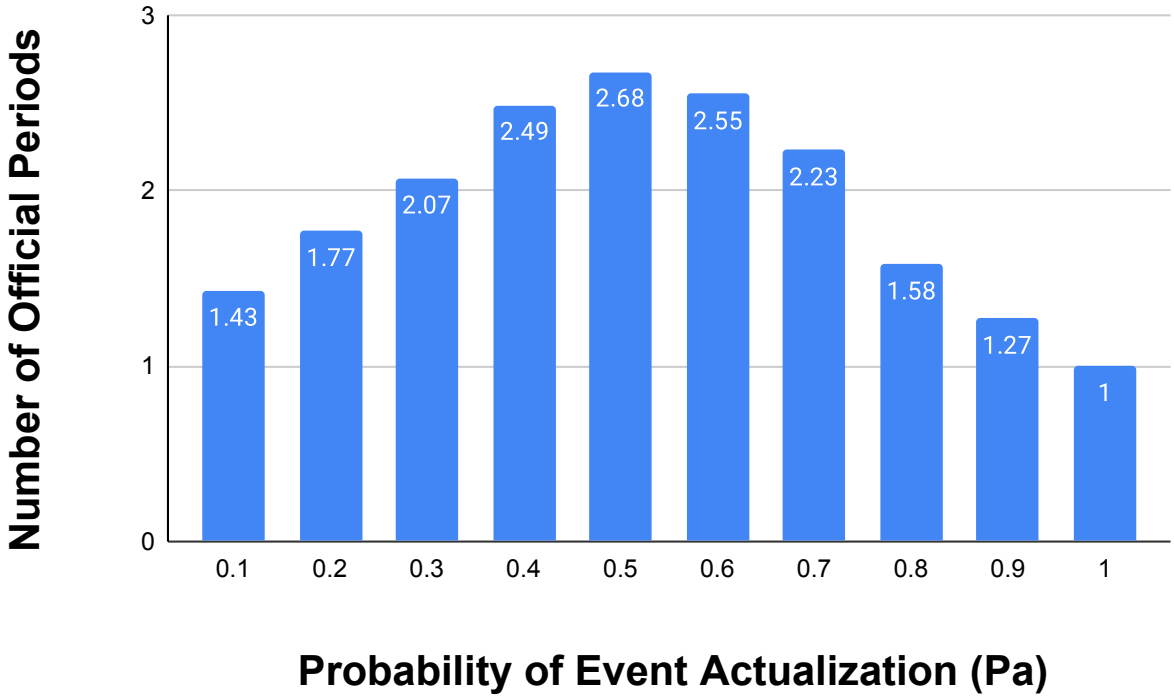
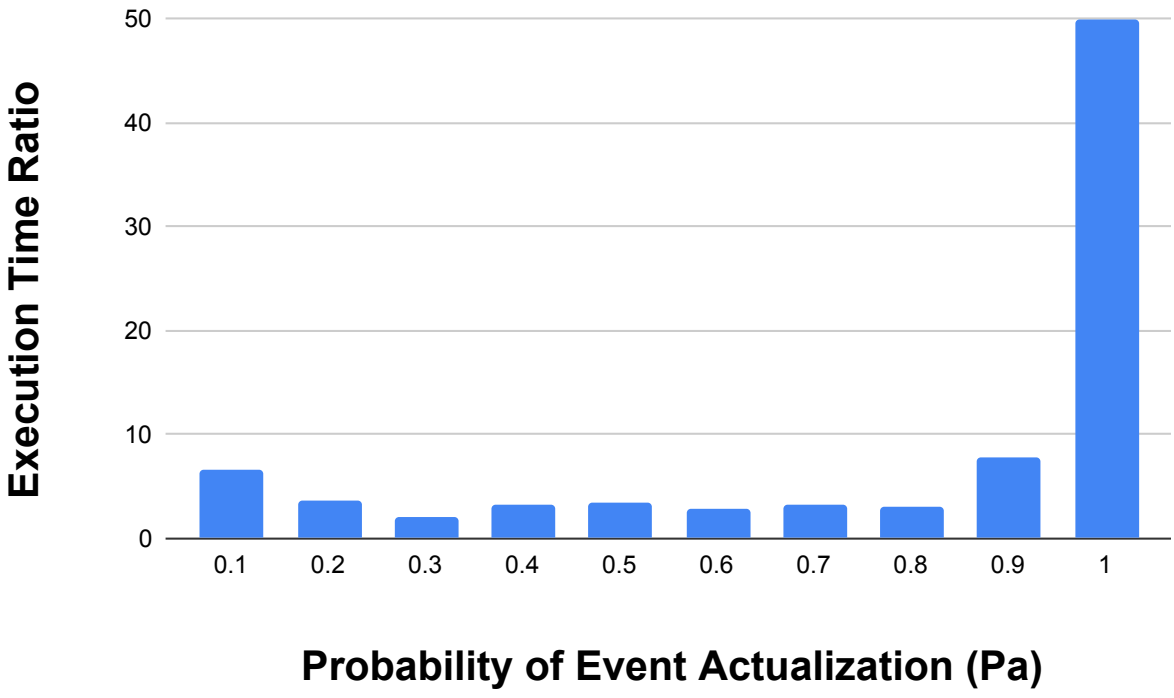


Figure 6.3: *FC-LBJ-CC* and *FC-LBJ Baseline* Execution Time Ratio



different values of P_a. The red bar shows the execution time for *FC-LBJ Baseline* and the blue bar show the execution time for *FC-LBJ-CC*. Observe that in case of *FC-LBJ Baseline*, as we increase the P_a the execution time first increases and then decreases. The same pattern can be observe for *FC-LBJ-CC* as well. If we look at the red or blue bar, we can observe that as the value of P_a increases, the execution time also increases. After a point, as the value of P_a increases, the execution time starts to decrease. We want to point out that when we look at the red bar (*FC-LBJ Baseline*), we are not observing a smooth increase and then a decrease in average execution time when P_a increases. This is because for not all execution complete within timeout and different values of P_a the number of successful executions are different. Whereas in the case of the blue bar (*FC-LBJ-CC*), we observe a smoother increase and then decrease in average execution time when P_a increases. This is because in *FC-LBJ-CC* when compared to *FC-LBJ Baseline* there were significantly fewer executions that were not completed within the timeout. The following explains this behaviour of first increasing and then decreasing execution time.

1. When P_a is small, what happens is that the history graph is very sparse. When we look for existence for official periods because it is overly constrained, they are pruning the search space very quickly. As a result, the search would fail very quickly.

2. As P_a increases, the pruning is not that effective. That is why it takes longer to find official periods. At these probabilities, we find the hardest instances.

3. As P_a increases further, what is happening is that the periods are becoming bigger, so it becomes easier to find the overlapping among periods and thus identify a solution. That is why the algorithm runs faster. At these probabilities, the graph is under-constrained.

The above observation is consistent with Figure 6.2. We can see that as P_a increases, the number of official periods increases, and then it start to decrease. For smaller values of P_a, there are fewer solutions found because the history graph is overly constrained. However, as P_a increases, the

history graph becomes less constrained; therefore, more official periods can be found. After a certain point, the graph becomes under constrained that the periods start merging and creating bigger periods. Therefore towards the end of the graph in Figure 6.2, we see a decrease in the number of official periods found.

Using the data from Figure 6.1 if we look at the blue bars, we can see the performance improvement we get by using *FC-LBJ-CC* instead of *FC-LBJ Baseline*. The performance improvement can be attributed to the reason that in *FC-LBJ Baseline*, we find all matches irrespective of whether the official period associated with certain matches is already found for the given inputs. Whereas in *FC-LBJ-CC* we stop the search for an official period for the match if the official period is contained in the already found official periods for the given inputs.

Figure 6.3, gives insights on the performance improvements we get from *FC-LBJ-CC* when compared to *FC-LBJ Baseline*. For each value of P_a we calculate execution time for *FC-LBJ Baseline* and *FC-LBJ-CC*. Suppose for $P_a = x$, the execution time for *FC-LBJ Baseline* is t_1 and for *FC-LBJ-CC* is t_2 then t_1/t_2 gives the ratio of the execution time of *FC-LBJ Baseline* and *FC-LBJ-CC*. Figure 6.3 plots these ratios of execution time of *FC-LBJ Baseline* and *FC-LBJ-CC* for different values of P_a. We observe that *FC-LBJ-CC* delivers better performance gains towards the start and end of the graph. From Figure 6.2 we can see that the start and end parts of the graph are where we find less number of official periods. So when there is less solution possible there the gains we get from *FC-LBJ-CC* are more. At P_a equal to 1, the performance improvement we get from *FC-LBJ-CC* is maximum. This happens because the periods for all timed-relationships in the history graph are the same and are equal to the timeline length T. So there is one possible solution for an official period that is equal to T. In *FC-LBJ Baseline*, when a solution is found, the algorithm searches for more matches, but *FC-LBJ-CC* utilizes the fact that any new match will be providing the same official period, and therefore *FC-LBJ-CC* using the containment check can skip the search for further matches.

In this section, we detailed the performance for finding official periods. To better understand

the overall performance of the HO(T)-ReBAC model, we need to investigate how quickly we can evaluate a HO(T)-ReBAC policy to get an authorization decision. In the next section, we detail the performance of policy evaluation in HO(T)-ReBAC.

6.3 Policy Evaluation

This section details the empirical study known as *Policy evaluation*. The motivation behind *Policy evaluation* is that we want to know the performance of policy evaluation in the HO(T)-ReBAC model. In *Policy evaluation*, we would be investigating the policy evaluation component of our HO(T)-ReBAC implementation. Section 6.3.1 details the experimental setup for *Policy evaluation* and in Section 6.3.2 we will present the measurements and results for *Policy evaluation*.

6.3.1 Experimental Setup For Policy Evaluation

For *Policy evaluation* we used the same hardware and software configuration as in *FC-LBJ-CC evaluation*. We stored the official periods found while running *FC-LBJ-CC* during the first part of the experiment in a relational database. We used MySQL [8] version 5.7 as the relational database for storing official periods. The number of stored official periods in the database is 1,047,311.

There are two data sets that we used for the *Policy evaluation* study. The first data set is based on the use case discussed in Chapter 4. The second data set is generated synthetically using the methods explained in the latter part of this section.

Use Case Policies: In Section 4.3 of Chapter 4, we described the real-world use case of HO(T)-ReBAC, which we called Medical IoT. There were three policy scenarios in Medical IoT, namely Real-time Device Monitoring, Remote Access, and Medical Data Gathering, for which we evaluated HO(T)-ReBAC. These Medical IoT policies have no more than 3 existential quantifiers. We also evaluated HO(T)-ReBAC for the g-SIS use case policies. There were 8 distinct policies in HO(T)-ReBAC that we used to encapsulate g-SIS as presented in Section 4.4 of Chapter 4. All these 8 policies had the same number of existential quantifiers, i.e., 2, therefore we presented the

average evaluation time for these 8 policies in Figure 6.4.

We evaluated each case-study policy against some randomly generated domain for each period variable appearing in the policy. (Recall that the domain of a period variable is simply a set of official periods.) Given a case-study policy ϕ, an ***instance*** of ϕ is the pairing of ϕ with a domain for each period variable appearing in ϕ. In *FC-LBJ-CC evaluation*, we computed $10 \times 1{,}000 = 10{,}000$ sets of official periods. We reuse these sets as domains for period variables. More specifically, a random instance of case-study policy ϕ is generated by assinging to each period variable of ϕ a domain randomly drawn from the 10,000 sets computed in *FC-LBJ-CC evaluation*.

Synthetic Policies: One of the parameters we look for in evaluation policies is the number of existential quantifiers in a policy. Increasing the number of quantifiers means increasing the period variables in a policy, thus introducing more checks required to evaluate a policy. The maximum number of quantifiers in the above-described use case policies is 3. Therefore we created synthetic policies using the HO(T)-ReBAC policy language with an increasing number of quantifiers in order to stress test *Policy evaluation*. The next part describes the method that we used to generate synthetic policies. We measured the evaluation time for different policies based on the number of existential quantifiers in a policy. We created these policies by varying the quantifiers count in the policies from 4 to 15. The following steps were used to synthetically generate a HO(T)-ReBAC policy:

1. Let n be the number of quantifiers in the policy, we first select n existential quantifiers with equal probability of selecting either $\exists$ or $\exists_\infty$ for each existential quantifier. We add the period variables associated with existential quantifiers in a period variable set, S_{pv}.

2. For each existential quantifier, we randomly chose a graph pattern from the database. Then we assign a period variable for each existential quantifier. This way we get n prefixes for the policy.

3. Then we generate the policy matrix by randomly selecting one of the five matrix

constructors (i.e., $\top$, *op*, $\neg$, $\wedge$, $\vee$) and then recursively generate subexpressions.

4. When the *I op J* construct is selected, we assign values for *I*, *J* and *op* as follows:

   (a) To select *I* and *J*, we randomly sample 2 variables from S_{pv}.

   (b) To select *op*, we randomly select an integer, *k*, between 1 and 13 (i.e., the total number of Allen relations). We randomly sample *k* relations from the set of 13 Allen relations.

Now that we have seen the setup for the *Policy evaluation*, in the next section, we will discuss the measurements and the empirical results for *Policy evaluation*.

6.3.2 Policy Evaluation Measurements and Results

In *Policy Evaluation* experiment, we are interested in the following two questions:

1. In Ho(T)-ReBAC, how efficient can we evaluate policies that are "naturally occurring"?

2. In Ho(T)-ReBAC, how efficient can we evaluate policies with a number of quantifiers larger than that in naturally occurring cases?

For the first question, we will measure the performance evaluation for the Medical IoT use case. We will stress test HO(T)-ReBAC using synthetic policies for the second question. In the following part, we will detail the measurements followed and the results we have for *Policy Evaluation* experiment.

Policy Evaluation Measurements: The experiments involve measuring the time for policy evaluation. We divided this into two parts. The first part involves measuring time for evaluation of policies that are based on the use case studies described in Chapter 4 as shown in Figure 6.4. The second part involves synthetic policies that are generated to measure the performance for stress testing the policy evaluation as seen Figure 6.5. We recorded 1,000 observations for each

Number of Existential Quantifiers	Successful Executions
4	1000
5	1000
6	1000
7	1000
8	1000
9	1000
10	1000
11	1000
12	1000
13	998
14	980
15	910

Table 6.2: Successful Synthetic Policy Executions

Figure 6.4: Use Case Policies Evaluation

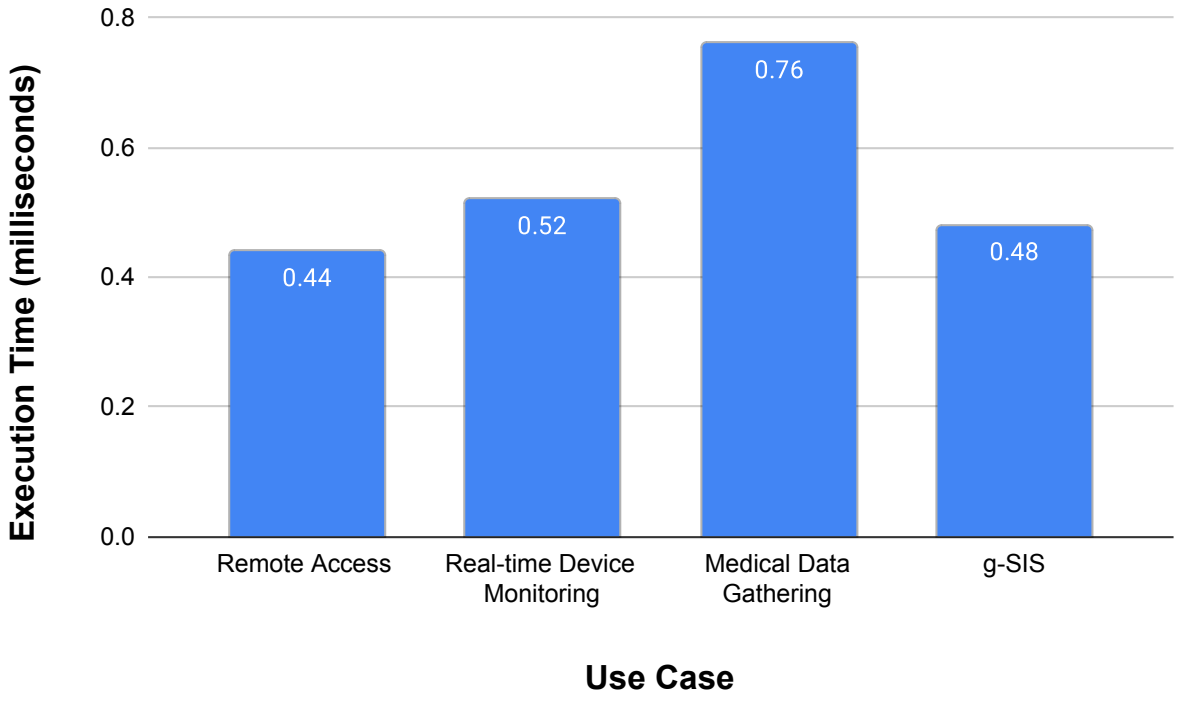
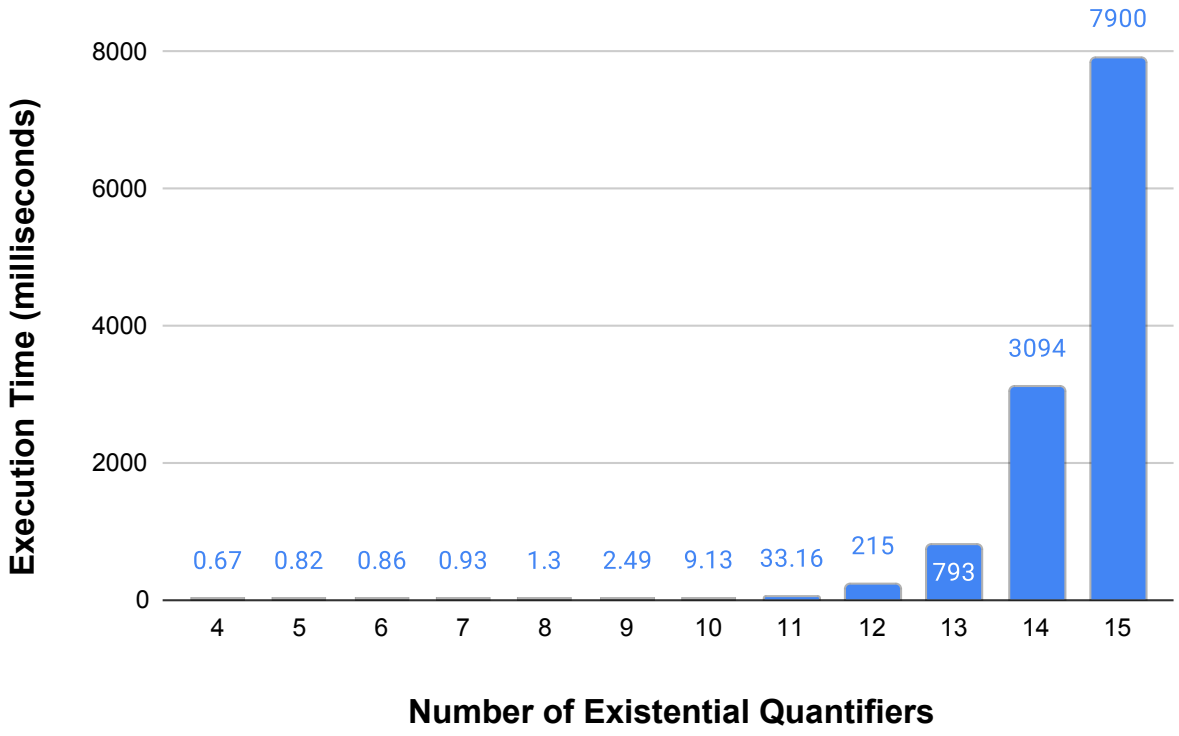


Figure 6.5: Synthetic Policies Evaluation



policy that was evaluated (both use case and synthetic). For the use case policies, we generated 1,000 policy instances for each policy. We evaluated each instance and computed the average evaluation time for the 1,000 instances. For the synthetic policies, for each number of quantifier, we generated 1,000 sample policies. We recorded time for evaluation for each and computed the average evaluation time for 1,000 trails.

We have used a timeout of 60,000 milliseconds for the measurements. We stopped that run and moved on to the subsequent execution for an execution that exceeded the timeout. When measuring the average time of execution we only used the executions that were completed within the timeout. For the use case policies, all executions were completed within the timeout. Table 6.2 shows the number of runs completed within the given timeout for synthetic policies. We can observe that all execution completed within the timeout in policies with up to 12 existential quantifiers. As we increase the number of existential quantifiers in a policy, the number of executions completed within the timeout reduces. This happens because the execution time increases when the number

of existential quantifiers in the policy increases. This behaviour can be observed in Figure 6.5 as well.

Figure 6.4 plots the policy name for the use-case policies on the X-axis and the execution time to get an authorization decision for the policy on the Y-axis. Whereas Figure 6.5 plots the number of existential quantifiers in a synthetically generated policy on the X-axis and the execution time to get an authorization decision for the policy on the Y-axis.

Policy Evaluation Results: We can observe Figure 6.4 and see the execution time for policies that were envisioned in our use cases described in Chapter 4. We can see that the maximum time for a policy in the case study takes 0.76 milliseconds. Therefore, since we have stored all the official periods required for a policy, getting an authorization decision is efficient (less than 1 millisecond). This supports the argument for HO(T)-ReBAC to be practical for at least the use cases we described in Chapter 4.

Now, we want to stress-test the efficiency of HO(T)-ReBAC by synthetically creating policies with an increasing number of existential quantifiers. In use case-based policies, the maximum existential quantifiers we have in a policy is 3. We scaled it up to 5 times, i.e., 15, for stress testing the number of existential quantifiers we observed in the use case policies. By observing Figure 6.5, we can see that for policies generated synthetically, as we increase the number of existential quantifiers, then the time to get an authorization decision increases. Increasing the existential quantifiers increases the conditions to be checked in a policy and increases the time to get an authorization decision. We can also observe that for policies with up to 13 existential quantifiers, we can get an authorization decision within 500 milliseconds (0.5 second). By increasing the number of existential quantifiers to 15, the execution time remains under 7900 milliseconds (7.9 seconds). The results for this stress test helps to support the argument that HO(T)-ReBAC is practical even with policies with a much greater number of existential quantifiers that we have envisioned in our use case.

Using the results from two studies *FC-LBJ-CC evaluation* and *Policy evaluation*, we argue that

HO(T)-ReBAC is efficient to find finding official periods and then finding authorization decisions using those official periods. This further helps in the argument for the real-world use of HO(T)-ReBAC. In the next chapter, we will describe the future work for HO(T)-ReBAC and the related works and conclude this thesis.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this work, we proposed an access control model, HO(T)-ReBAC, that accounts for transient relationships in a ReBAC model. HO(T)-ReBAC is designed for large-scale real-world applications with transient relationships between entities such as users, resources, or groups. We examined the usefulness of HO(T)-ReBAC in the real world. We formalized the HO(T)-ReBAC model and defined a formal policy language specifying authorization conditions. We then explored the type of transient relations that can be found in the real world. We also examined a concrete example of a smart hospital setting where HO(T)-ReBAC can be deployed for making authorization decisions. We further investigated how HO(T)-ReBAC can be used in group settings by emulating the group operations of g-SIS.

To evaluate HO(T)-ReBAC policies, official periods are required. For efficient authorization decisions, we have described the problem of finding official periods as a constraint satisfaction problem. We presented an efficient algorithm called *FC-LBJ-CC* to find official periods. We then presented a method to cache official periods in a database. After this, we described a translation from HO(T)-ReBAC policy to database query to compute authorization decisions. We empirically evaluated the performance of HO(T)-ReBAC by dividing the evaluation into two parts. First, we evaluated the performance of *FC-LBJ-CC*, and after that, we evaluated the performance of the database queries we generated to find authorization decisions.

We argued for the use of HO(T)-ReBAC for real-world applications because (i) there is a formalization present for HO(T)-ReBAC model, (ii) a policy language for HO(T)-ReBAC, (iii) a study that investigates the real-world use cases for HO(T)-ReBAC, and (iv) an empirical evaluation of HO(T)-ReBAC that found it to be efficient to be used in the real world.

7.2 Future Work

There are several ways to extend this work, leading to number of research opportunities.

1. Extending the model to incorporate the attributes for the entities such as users, resources, and groups. As shown by [30], ReBAC and ABAC can be combined, and the future work can include the exploration for combining ABAC with HO(T)-ReBAC.

2. In our work, we considered temporal constraints among the relationships. In the future, this temporal constraint can be replaced with spatial constraints for the relationships. In such a model, the authorization decisions will be based on how the locations of the entities are related to each other. The new model can be called *Higher-Order (Spatial) Relationship-Based Access Control (HO(S)-ReBAC)* model.

3. A further research opportunity would be to introduce a generic Higher-Order ReBAC model that can incorporate in one model various constraints like temporal, spatial, or any other constraints considered necessary in the future. The new model can be called *Higher-Order Relationship-Based Access Control (HO-ReBAC)* model.

4. We explored the use of HO(T)-ReBAC in the healthcare industry. In the future, other industries with transient relations can be explored where HO(T)-ReBAC can find usage. One such industry can be autonomous vehicles, in which there are transient relationships between nearby vehicles during some period(s) that can be used to decide whether to share information or not.

5. Further exploration in caching the official periods can be done and compared in future work. This can help further improve the efficiency of HO(T)-ReBAC.

Bibliography

[1] Allen's Interval Algebra. https://www.ics.uci.edu/ alspaugh/cls/shr/allen.html, 2022.

[2] Amazon Web Services EC2 Instance. https://aws.amazon.com/ec2/instance-types/, 2022.

[3] Apple Watch. https://www.apple.com/ca/watch/, 2022.

[4] AReBAC Implementation. https://github.com/szrrizvi/arebac, 2022.

[5] Cypher Query Language. https://neo4j.com/developer/cypher-query-language/, 2022.

[6] Fitbit. https://www.fitbit.com/en-ca/home, 2022.

[7] MS Office 365. https://www.microsoft.com/en-ca/microsoft-365, 2022.

[8] MySQL. https://www.mysql.com/, 2022.

[9] Neo4j. https://neo4j.com/, 2022.

[10] Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html, 2022.

[11] Whatsapp. https://www.whatsapp.com/, 2022.

[12] ALLEN, J. Maintaining knowledge about temporal intervals. *Communications of the ACM 26*, 11 (1983), 832–843.

[13] BERTINO, E., BONATTI, P. A., AND FERRARI, E. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur. 4*, 3 (Aug. 2001), 191–233.

[14] CARMINATI, B., AND FERRARI, E. Enforcing relationships privacy through collaborative access control in web-based social networks. In *Proceedings of the 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'09) (Washington DC, USA, Nov. 2009)*, pp. 1–9.

[15] FONG, P. W. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004* (2004), IEEE, pp. 43–55.

[16] FONG, P. W., MEHREGAN, P., AND KRISHNAN, R. Relational abstraction in community-based secure collaboration. In *Proceedings of the 20th ACM Conference on Computer and Communications Security* (Berlin, Germany, 2013), CCS '2013, ACM, pp. 585–598.

[17] FONG, P. W. L. Relationship-based access control: protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy* (San Antonio, TX, USA, 2011), CODASPY '2011, pp. 191–202.

[18] GATES, C. E. Access control requirements for web 2.0 security and privacy. In *IEEE Web 2.0 privacy and security workshop (W2SP'07)* (2007).

[19] HELLY, E. Über mengen konvexer körper mit gemeinschaftlichen punkte. *Jahresbericht der Deutschen Mathematiker-Vereinigung 32* (1923), 175–176.

[20] HU, V. C., FERRAIOLO, D., KUHN, R., SCHNITZER, A., SANDLIN, K., MILLER, R., AND SCARFONE, K. Guide to attribute based access control (ABAC) definition and considerations. NIST Special Publication.

[21] JOSHI, J., BERTINO, E., AND GHAFOOR, A. Temporal hierarchies and inheritance semantics for GTRBAC. In *Proceedings of the seventh ACM symposium on access control models and technologies* (2002), SACMAT '02, ACM, pp. 74–83.

[22] JOSHI, J., BERTINO, E., LATIF, U., AND GHAFOOR, A. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering 17*, 1 (2005), 4–23.

[23] KRISHNAN, R., NIU, J., SANDHU, R., AND H. WINSBOROUGH, W. Group-centric secure information-sharing models for isolated groups. *ACM Transactions on Information System Security, 14*, 3 (2011).

[24] PEREIRA, H. G. G., AND FONG, P. W. L. SEPD: An Access Control Model for Resource Sharing in an IoT Environment. In *Computer Security – ESORICS 2019* (Cham, 2019), K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., Springer International Publishing, pp. 195–216.

[25] PROSSER, P. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence 9*, 3 (1993), 268–299.

[26] R. RIZVI, S. Z., AND FONG, P. W. L. Efficient authorization of graph database queries in an attribute-supporting ReBAC model. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA, Mar. 2018), CODASPY '2018, pp. 204–211.

[27] RICHARDSON, M., AGRAWAL, R., AND DOMINGOS, P. Trust Management for the Semantic Web. *ISWC* (2003).

[28] RIZVI, S. Z. R., AND FONG, P. W. L. Efficient authorization of graph database queries in an attribute-supporting rebac model. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA, 2018), CODASPY '18, p. 204–211.

[29] RIZVI, S. Z. R., AND FONG, P. W. L. Efficient authorization of graph-database queries in an attribute-supporting rebac model. *ACM Trans. Priv. Secur. 23*, 4 (jul 2020).

[30] RIZVI, S. Z. R., FONG, P. W. L., CRAMPTON, J., AND SELLWOOD, J. Relationship-based access control for an open-source medical records system. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies* (Vienna, Austria, Mar. 2015), SACMAT '2015, p. 113–124.

[31] ROSSI, F., BEEK, P. V., AND WALSH, T. Handbook of constraint programming. *Elsevier* (2007).

[32] SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. Role-based access control models. *Computer (Long Beach, Calif.) 29*, 2 (1996), 38–47.

[33] SANDHU, R. S. Role-based access control. In *Advances in computers*, vol. 46. Elsevier, 1998, pp. 237–286.

[34] TANDON, L., FONG, P. W., AND SAFAVI-NAINI, R. HCAP: a history-based capability system for IoT devices. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies* (2018), pp. 247–258.