Foundations and Implementations of Declarative Access Control for Online Social Networks

Edward Caprin

A thesis submitted for the degree of
Doctor of Philosophy
at
Western Sydney University

May 2016

Dedication

To my family.

Acknowledgements

I begin by thanking my primary supervisor; Professor Yan Zhang. Without him this research would have not been possible. As my mentor throughout my Honours and Ph.D. studies he has provided me with invaluable guidance and support. The lessons and skills Yan has taught me will always be of value in my future endeavours. For this I will always be grateful.

I thank the my co-supervisor Dr. Yi Zhou along with members of the Artificial Intelligence Research Group, past and present. They have all contributed to my understanding of the my own field and have exposed me to many other exciting areas.

Thanks also to the other staff and research students of the School of Computing, Engineering and Mathematics. During my time within the School they have provided me with valuable advice and perspective. Through fun stories and anecdotes about their own Ph.D. experiences I have been able to better gauge and appreciate the progress I have made.

Thank you to the Western Sydney University for providing the facilities and resources necessary to undertake my research through the Australian Postgraduate Award and Top-Up Scholarship.

I finally thank my father for his many years of care, advice, and support. Without him I would have not been able to reach many of my achievements throughout my life. His unwavering support in all of my endeavours and interests has made me who I am today, and for this I will always be grateful.

The work presented in this thesis is, to the best of my knowledge, original except as acknowledged in the text.

I declare that I have not submitted this material, either in full or in part, for a degree at this or any other institution.

. . . .

Edward Caprin May 30, 2016

Contents

List of Figures

Abstract

In a relatively short period of time Online Social Networks (OSNs) have become an integral part of many people's lives. They provide an easy to use environment for keeping in touch with family and friends, sharing content such as photos, and organising events. More often than not to fully utilise an OSN, users are required to disclose personal information. For instance, when setting up a new Facebook account new users need to provide a first and last name, email address, and their date of birth.

Unsurprisingly, the widespread disclosure of personal information has led to growing concerns about OSN privacy management amongst academia, OSN users, and the wider community. Much of the concern focuses on the unintentional or inadvertent disclosure of one's personal information to unexpected parties. For example, a private photo of an OSN user at a wild party being unknowingly shared with their boss or coworkers. In this scenario the disclosure results in embarrassment for the user and potentially had a negative influence on their employer. Given in more serious instances an unwanted disclosure could lead to identity theft and, in extreme cases, physical harm it is important that they are addressed.

In this research OSN privacy management is approached as an access control problem by proposing an Attribute-Based Access Control (ABAC) framework tailored to OSNs. This basis on the emerging model ABAC allows for the use of the wide assortment of security relevant information already present in OSNs when devising a user's access policies. Furthermore, this research performs a formal investigation of the challenges presented by the expression of, reasoning with, and update of ABAC policies. Through these investigations this research has developed formal foundations and implementations for each of these key facets of ABAC.

The first of these foundations is the ABAC policy specification language SocACL. With features tailored to OSNs and semantics defined as a translation from SocACL to Answer Set Programming (ASP) the language allows for the application of logic programming techniques and research to aspects of OSN privacy management. By leveraging SocACL's ASP semantics, the language is supported by our proposed policy evaluation system based on the novel application of negotiations.

Since at some point a user's SocACL or ABAC policies will need to be updated to reflect their ever changing privacy preferences, we have also developed a formal ABAC policy update methodology. This methodology considers OSN policy updates as reactionary, allowing for the user to define the update request as a set of observed, but, unwanted access control outcomes. Similar to our negotiation based policy evaluation, this policy update adopts techniques originally developed for logic programming. Each of these foundations is supported by a prototype implementation which makes use of ASP solvers to perform key computations.

This thesis describes both the foundations and implementations of our OSN privacy management system comprised of ABAC policy expression, evaluation, and update formalisms. These formalisms are presented and analysed in their respective chapters. We also provide a technical overview of their implementations and discuss various case studies, experiments, and performance results.

List of Abbreviations

ABAC	Attribute-Based Access Control
ABE	Attribute-Based Encryption
ASP	Answer Set Programming
ATN	Automated Trust Negotiation
DAC	Discretionary Access Control
DL	Delegation Logic
DMS	Dynamic Magic Sets
EBNF	Extended Backus-Naur Form
FOL	First Order Logic
FtG	Friend-to-Group
MAC	Mandatory Access Control
MPAC	Multiparty Access Control
MSM	Mainstream Media
NAF	Negation as Failure
NIST	National Institute of Standards and Technology
NKB	Negotiation Knowledge Base
NLP	Normal Logic Program
OSN	Online Social Network
P-RBAC	Privacy-Aware Role-Based Access Control
PA	Policy Authority

PB	Policy Base
PDO	PHP Data Objects
PKE	Public Key Encryption
RBAC	Role-Based Access Control
ReBAC	Relationship-Based Access Control
SL	Sensitivity Level
SP-RBAC	Sharing and Privacy-Aware Role-Based Access Control
SoD	Separation of Duty
TM	Trust Management
UC	Update Candidate
UIV	Update Impact Value
UP	Update Program
UR	Update Request
XACML	Extensible Access Control Mark-up Language

Chapter 1

Introduction

1.1 Introduction and Motivation

In a relatively short period of time Online Social Networking services, such as Facebook and LinkedIn, have become an integral part of peoples' lives. They provide an easy to use environment for keeping in contact with family and friends, sharing media, and forming communities.

These services often require users to disclosure significant amounts of personal information to properly participate in the network. For example, setting up a new Facebook account requires a user's full name, email, and date of birth. Following this initial set-up is the user is then encouraged to provide additional information, such as the educational institutions they have attended or his/her workplace.

This information is used by the service to assist in finding other members of the Online Social Network (OSN) the user may be interested in becoming "Friends" with. This "Friend"-ship denotes a link between the profile pages' of users and does not imply the existence of any out-of-network relationship. For convenience we distinguish between OSN "Friends" and out-of-network "friends" by capitalising the former.

Unsurprisingly, such disclosure of personal information has resulted in OSN privacy issues becoming of interest amongst academia [39, 64, 65] and in the general population through Mainstream Media (MSM) reporting [68, 82]. OSNs are particularly vulnerable to privacy breach attacks [39]. In these attacks a user's private information is accessed by unwanted individuals. The severity of these attacks varies greatly. They can result in the embarrassment of the user. Be used as a channel for identity theft. Negatively influence the decision of a potential employer. Or in the most extreme cases, result in physical harm.

To their credit, operators of many high profile OSNs have responded to user's privacy concerns. For instance, changes in Facebook's API, Graph, from version

1.X to 2.0 made it more difficult for applications to access a user's Friends list. OSNs also often provide user definable privacy settings where Friends are mapped to sharing permissions. However these have been ineffective and often result in configurations which do not reflect the user's privacy intentions [64, 65].

The research presented in this thesis approaches OSN privacy management as an access control problem. To this end, we have developed an implementation of the Attribute-Based Access Control (ABAC) model based on Answer Set Programming (ASP) for privacy management in OSNs. This development has centred around the formal investigation of three core areas; *policy expression*, *policy evaluation*, and *policy update*.

1.1.1 Policy Expression

The goal of any access control scheme is to limit who can access particular resources. In order to form rational access control decisions there must be some source of information on which to base them. Access permissions are mapped to a set of decision criteria to form *rules*. Sets of these rules form *Policies* which encode a user's access control preferences. Policy expression concerns itself with the syntax and semantics of such rules.

1.1.2 Policy Evaluation

When a user is presented with a request for his/her resources they make a decision on whether to allow or deny access. This decision is based on the user's access control preferences encoded by the semantics of his/her policy. The process of determining the outcome of a request with respect to a policy is called policy evaluation.

1.1.3 Policy Update

At some point the circumstances which the user has based their policy on will change. In turn, the policy must be adjusted, or updated to reflect this change. This alteration of a policy with respect to some change is called a policy update.

1.1.4 Internal Thesis Referencing

Throughout this thesis there are a number of internal references to other sections of the thesis and the definitions, examples, figures, etc. they contain. As these references are important to understanding the research presented we clarify the reference convention used for the duration of this thesis.

Chapter, Sections and Subsections

References to chapters, sections, and subsections (including the appendices) follow a 3-block system. For a reference of the form *C.S.SS*, *C* denotes the chapter, *S* is a section within Chapter *C*, while *SS* is a subsection of Section *S*. For example:

- Chapter 1: the first chapter of this thesis.
- Section 1.2: Section 2 of Chapter 1.
- Section 1.2.3: Subsection 3 of Section 2 of Chapter 1.
- Appendix A.1.2: Subsection 2 of Section 1 of Appendix A.

Examples, Equations, Figures, etc.

Sections contain examples, equations, figures, etc. These are collectively referred to as *environments*. Unlike section references, environment references follow a 2-block convention where the first digit is the chapter where environment appears and the second digit is denotes its sequential position in the chapter. Algorithm environments are the only exception to this by following a single digit system. For example:

- Equation (2.2): 2nd equation found in Chapter 2.
- (2.2): Alternative formatting for the 2nd equation found in Chapter 2.
- Example 4.9: Example 9 of Chapter 4.
- Definition 4.9: Definition 9 of Chapter 4, each environment has a distinct sequence count.
- Algorithm 4: 4th algorithm in this thesis.

1.2 Background Knowledge

As with any literature review the one presented here aims to provide background into the research topic and investigate related work. To do this we explore texts on three broad topics; OSNs, Access Control, and ASP.

The review begins with a introduction to OSNs by providing a background and characterisation of these services. After which is a presentation of the threats faced by OSNs and studies into the effacy of mitigation strategies employed by popular OSNs. This is followed by an overview of privacy management approaches proposed by academia.

Given this research considers OSN privacy management as an access control problem the review explores access control. This considers attempts the apply the popular Role-Based Access Control (RBAC) model to OSNs as well as models developed specifically for OSNs. This coverage on access control concludes with a discussion of ABAC, the current research gap surrounding the model, and justification for why this model was selected as the focus for this research.

The final section of this review provides insight into logic programming, ASP, and how these technologies can be applied to the challenges presented by ABAC policy expression, evaluation, and update.

1.2.1 Online Social Networks

Characterising Online Social Networks

In part the success and popularity of OSNs can be credited to their targeting of specific audiences. For example, Last.FM is tailored to promote interactions between people of similar musical taste, while LinkedIn focuses on professional networks. There are also examples of early OSNs which catered to specific ethnic communities, such as AsianAvenue, MiGente, and BlackPlanet [18].

This variety is excellent for the discerning OSN user, but it represents a challenge for developing a working definition for OSNs. Given the wide spread use this thesis adopts of the definition proposed by Boyd and Ellison [18], where an OSN is a web-based service which allows individuals to:

1. Construct a public or semipublic profile within a bounded system,
2. Articulate a list of other users with whom they share a connection, and
3. View and traverse their list of connections and those of others within the system.

This definition focuses on the ubiquitous OSN Friend relationship. Friend, or variation of, relationships often aim to model real world social connections within the OSN. As it intuitively follows people like to share "stuff" with Friends, OSNs often base their privacy management settings on them. However the literature notes this is problematic [39, 64, 65].

Threats to OSNs

In their 2011 study Gao et al. [39] summarise security threats to OSNs. Gao et al, categorise these threats into four different types: *Privacy Breach Attacks*, *Viral Marketing*, *Network Structural Attacks*, and *Malware Attacks*.

In a privacy breach attack the personal information of a user is revealed to an entity who would not normally have access. OSNs are particularly vulnerable to these attacks as they encourage their users to provide astonishing amounts of personal information. To illustrate how much information is voluntarily provided to OSNs Gross and Acquisti [48] studied Facebook users at Carnegie Mellon University finding that:

- 90.8% of users uploaded an image to Facebook.
- 87.8% display their date of birth.
- 39.9% their phone number.
- 50.8% of user's profile pages listed their current residence.

In response to their user's concerns about privacy many OSNs provide user configurable privacy settings. However, the inclusion of such settings has done little to improve user's privacy outcomes. This is largely because the task of configuring these settings is too difficult for many users [64, 65].

Even for the professionally trained the authoring of access control policies is difficult and time consuming. In OSNs users have the expectation to be able to set their own privacy settings despite the complexity of the environment. So it is of little surprise that studies into how users configure their privacy settings in Facebook, such as work by Lipford et al. [64] and Madejseki et al. [65], conclude that users generally devise poor policies. The results of [65] are of particular interest because they consider unintentional hiding of information as errors, rather than what is normally associated with setting errors; unintentional exposure of information.

Madejski et al. [65] observed that overwhelmingly users' privacy settings did not align with his/her sharing intentions and that every participant in the study, 65 in total, had at least one incorrect setting. When presented with the error the 87% of participants stated they could not or would not correct the error.

Participants in the "could not" group suggests the feedback provided by the system is either insufficient or too difficult to understand the cause of the error. The results of Cheek et al. [24] and Lipford et al. [64] supports this theory. Both of these studies developed prototype systems for Facebook focussing on improving how feedback on privacy settings is presented to the user. They both observed significant improvements in the accuracy of settings when participants used their respective prototype. These prototype systems are discussed in further detail later in this review.

Participants stating they "would not" correct the error are more difficult to understand. It could simply be a case of laziness, with the respondents not considering the error significant enough to warrant a response. Another possibility is that they have similar issues to the "could not" respondents, but have chosen to given a sterner response. Unfortunately Madejseki et al. do not provide any further comments on this.

OSNs attempt to model social structures by linking profile pages through the Friend relationship. In many OSNs these relationships are the basis of the network's privacy settings with simple rules such as "only let Friends view this" and "can be viewed by public". This concept of Friend has been criticised as being insufficient for capturing the complexity and nuance of real world relationships. Gao et al. [39] also identify this as a problem explaining that:

> ...the notion of 'friends' in an OSN is merely a social link that the two users have agreed to establish in that OSN, regardless of the actual offline relationship.

This lack of fine-grained relationship handling can lead to a number of issues. Policies based around these relationships are inherently flawed as they do not reflect the level of trust the user wants to convey. Furthermore, such simple relationships leave users vulnerable to even basic attacks, as noted by [39]. One solution to this would be to increase the types of relationships available in the framework, but this still limits the framework to the relationship types it was designed with. A better solution would to allow for the OSN users or the OSN service provider to define their own relationship types.

Gao et al. also discuss the threats of viral marketing, network structural attacks, and Malware. However, as the focus of this thesis is privacy management as an access control problem they are outside of the scope of this review.

Approaches from Academia

In response to the issues highlighted in the previous section the research community has proposed a variety of novel approaches to OSN privacy management. Privacy settings currently offered by OSNs allow for the assignment of users defined Friend lists to permissions. Facebook provides similar functionality with their "Lists" feature, Google+ calls these groups "Circles".

Due to the potentially large number of Friends the task of assigning them groups can be daunting. Cheek et al. [24] ascribes this difficulty to users having to change their mental focus, or mental modes, many times during the process of Friend-to-Group (FtG) assignment. They [24] explain when a user is presented with a Friend they have to consider details about the Friend then consider which group they best fit into, while also considering characteristics of the group. This is repeated with potentially hundreds of Friends with each Friend and group requiring a change in mental mode. Cheek et al. hypothesise that by minimising the number of changes to a user's mental mode they can make FtG assignment easier and faster. Cheek et al. [24] present two different approaches to achieving this.

Their first approach is *Assisted Friend Grouping.* This model utilises well known clustering techniques [24] to order a user's Friends. Friends are ordered by an algorithm, such that Friends in close proximity to each other in the list are believed to be in the same group. During the FtG assignment process Friends are presented to the user in this order. Cheek et al. believe by doing this the number of changes to the user's mental mode is reduced since the user now considers the same group for many Friends. One of the dangers of this sort of system is for it to overly influence the user's decision process, potentially leading to security problems due to incorrect assignments or bias in the algorithm. Assisted friend grouping mitigates this by acting as a recommender only, and not performing the actual assignment. This approach relies on the user's own judgement overruling "odd" recommendations.

Their second model, *Same-As Policy Management*, "...leverages their [the user's] memory and opinion of a Friend to set policies for other like Friends". In this approach the user nominates an example Friend which is representative of a subset of the user's Friends list. Using a visual policy editor the user then assigns access permissions to this example Friend. Friends which have been algorithmically determined to be similar to the example are then assigned the same permissions. This model is interesting for a number of reasons. Firstly, it utilises a visual editor that provides clear and immediate feedback about the policy settings. Secondly, it leverages the natural behaviour of treating Friends that are similar in similar ways. Thirdly, this model suggests the ability to assign permissions in the absence of knowing the Friends identity, allowing for the dynamic assignment permissions to strangers.

The work presented by Cheek et al. [24] focuses on improving systems already in place in OSNs. On the complete opposite end of the spectrum is the work by Baden et al. [7] where they present an decentralised OSN framework based around Attribute-Based Encryption (ABE) and traditional Public Key Encryption (PKE). This framework, named Persona, hides all information from the service provider and other users by using cryptographic techniques. Persona can be seen as the logical extreme of privacy preservation solutions for OSNs.

In Persona's framework users create groups using arbitrary criteria, though Baden et al. assume users will choose transparent relationships such as "coworker". Data is encrypted with respect to these group criteria, effectively restricting access to the data to the specified groups. Baden et al. specify two options for cryptographic approaches; PKE and ABE. The PKE approach follows traditional public key and symmetric cryptography, distributing keys when communications occur. The ABE approach encrypts data using keys generated from logical expressions of attributes, such as “neighbour” AND “football fan”. Baden et al. [7] comment this approach is better suited to OSNs than traditional PKE since it exploits information already in the OSN. By hiding all information and being decentralised Persona addresses many of the issues identified by [39], specifically, the inherent trust users must place in the service provider.

However, this results in a system which even Baden et al. themselves admit has little incentive to ever be implemented because there is no viable business model to support it. The popularity of OSNs partially relies on them being free. Currently popular OSNs leverage huge quantities of user data to generate income through various schemes. Persona hides all user information unless the user explicitly desires otherwise. Thus eliminating any business model that exploits user data. The other option is a pay-to-use or subscription system. After the community backlash at unsubstantiated rumours of Facebook changing to a pay-to-use model [13] it is unlikely that any OSN would be successful under such a model. As such, Persona stands an interesting example of a privacy centric OSN framework and little more. It is an example of the logical extreme of privacy preservation strategies for OSNs, highlighting important considerations for our own system. The system has to strike a balance between privacy preservation and business soundness otherwise it would never be implemented. Secondly, we are again presented with a privacy management solution tailored to OSNs with a strong focus on attributes.

1.2.2 Access Control

Access control is fundamental to information security. It ensures the integrity, confidentiality and availability of information. As this thesis approaches OSN privacy management as an access control problem it is important to consider which model to used. In this subsection we provide an overview of various access control models and their application to our problem domain.

We begin by exploring the immensely popular RBAC model and the attempts at applying it to OSN privacy management. Following this is an overview of access control models specifically designed for OSNs. Finally this review of access control concludes with literature on the model this research focuses; ABAC.

Role-Based Access Control (RBAC)

RBAC is a popular access control model introduced by Ferraiolo et al. [34]. RBAC was developed to address the perceived inadequacies of the Mandatory Access Control (MAC) and Discretionary Access Control (DAC) models. At the time Ferraiolo et al. took the firm position that reliance on DAC as the principal access control model is "unfounded and inappropriate for many commercial and civilian government organisations" [34].

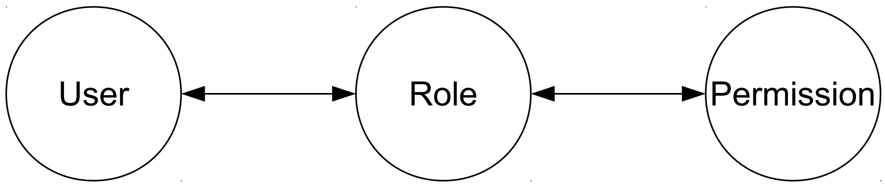


Figure 1.1: RBAC User-Role-Permission Mapping

Normally individuals within an organisation fulfil a specific function. More often than not, in order to carry out this function these individuals have restrictions on what they can and cannot do. For instance, in a hospital doctors can prescribe medicine, while cleaners cannot. RBAC takes advantage of this organisational feature by assigning permissions to roles which represent a job or function. Roles are then naturally assigned to individuals by their job description. This User-Role-Permission mapping is illustrated in Figure 1.1.

Since first proposed RBAC has proven incredibly popular in both academia [27, 63, 69] and industry with large software vendors such as Microsoft and Oracle providing RBAC implementations in their products. RBAC has become so ingrained in information security that it is effectively the *de facto* standard for access control. As a result there is a strong trend of applying RBAC to a wide range of access control domains, including OSN privacy management. However, it is becoming increasingly evident that the assumptions core to RBAC do not hold in modern web environments.

RBAC relies on two key assumptions 1) user-role assignments change infrequently and 2) that roles are well defined [34]. Though these assumptions hold in the majority of organisations they are at odds with OSNs. For an OSN it is difficult to assume users change infrequently as it runs counter to user behaviour. User's can change their profile page along with its associated content and relationships any time they wish. Additionally, users can leave or rejoin the network at any time. This clearly conflicts with assumption 1).

The issue with assumption 2) is that OSN users do not hold a well defined role within the network. At a high network level, some users will be prolific contributors of content, others will "lurk", while some users will simply stop engaging with their account. Even if roles are considered at a "local" Friend level roles are still unclear because these Friend-like relationships can change rapidly to reflect offline interactions. For this reason many researchers have proposed extensions or "flavours" of RBAC which reconsider assumptions 1) and 2) when applied to different domains.

One of these is Privacy-Aware Role-Based Access Control (P-RBAC) proposed by Ni et al. [69]. P-RBAC aims to improve RBACs support for privacy policies by integrating the key concepts of *purpose* and *obligations*. These concepts are core to the *OECD Guidelines on the protection of privacy and transborder flows of personal information* [70]. Purpose stipulates that personal information is only used for the reason it was accessed, for example an email provided for sending the customer an invoice cannot be used for third-party advertising. Obligations are an agreement between the resource owner and requester that the later performs a specified action after accessing the resource they requested.

Continuing the trend of extending RBAC, Malik et al. [66] propose Sharing and Privacy-Aware Role-Based Access Control (SP-RBAC). SP-RBAC extends NIST RBAC to address privacy concerns in OSNs. This model addresses the lack of well defined roles in OSNs by reinterpreting them. RBAC roles are replaced with *collaborative groups*; user defined groups which represent some social circle or group, such as a club, school or workplace. Permissions are then mapped to these groups. User-to-collaborative group assignment is performed at the discretion of the of resource owner or by some dynamic process.

Collaborative groups go beyond simply renaming roles through the inclusion of another novel feature that aims to model trust variations within social circles. Intuitively within a social group we do not treat or trust everyone equally. Trust is further differentiated between the members based on emotional relationships. For instance, a member of the "Genericville Chess Club" circle may only be friends with some of the members, and dislike others. To take these emotional relationships into account Malik et al. [66] introduce *collaborative relationships*. Collaborative relationships are levels of emotional trust assigned to individuals within a certain collaborative group. This adds both an additional layer of specification precision to the model and makes it more comparable to how people handle real world relationships. This is interesting as it indicates that it is possible to use the structure of RBAC in ways not originally intended. Furthermore, by doing so it is possible to tailor new models to a specific problem domain, in this case OSNs.

Besides extending existing models, as the above examples have done, research has been conducted with the aim of developing new access control models specifically for modern web services, such as OSNs.

OSN Specific Models

With relationships playing an integral role in any OSN [18] it should come as little surprise that access control models specifically for OSNs leverage them [30, 37, 50, 66]. Dhia [30] propose a framework based on node reachability by using user relationships as edges and profile pages as nodes. By mapping permissions to sequences of relationships they reduce policy evaluation to finding paths in the social graph.

Relationship-Based Access Control (ReBAC) proposed by Fong et al. [37] describes access permissions in terms of the accessors relationship with the owner. These relationships can be inverted to derive the opposite direction, implying all relationships are bidirectional, e.g. inverse of a "parent" relationship is "child". Primitive relationships, such as "parent", can be combined to form complex ones, "parent parent" is a "grandparent". However, this results in ambiguity as to whether or not "grandparent" should be considered a 1st- or 2nd-degree relationship.

OSNs often allow users to upload content, such as photos, to the service to allow for easy sharing with Friends. Typically the user who uploads the content is considered the owner of said content by the OSN, with the owner being given full control over how the content is shared. As noted by Hu et al. [50] this shared content can have multiple stakeholders. For example, for a photo of friends at a party, everyone that appears in that photo has a stake in how it is shared, but only the upload/owner has control over its sharing in the OSN. Hu et al. call this sort of scenario a "multiparty" access. They [50] argue the privacy settings, at the time, provided by OSNs do no adequately cater for multiparty access. To address this they propose a new access control model specifically for OSN multiparty access; Multiparty Access Control (MPAC). MPAC is interesting due to it being based on logic programming and its novel policy evaluation system.

Differing from both Dhia's graph-based approach [30] and ReBAC, in MPAC an access permission is defined as a construct called a *policy*. The policy identifies the authority defining the policy, what stake they hold in the content, identifiers for the content accessors, and a identifier of the content itself along with a Sensitivity Level (SL). Each MPAC policy is defined w.r.t. a policy authority which has some stake in the content. Accessors can be identified by either their name, group, or relationship with the policy authority. The content the policy applies to is identified along with a SL. SL's are used with MPAC's most interesting feature; vote-based policy evaluation.

Key to MPAC addressing access to multiparty assets is to allow for the collaborative evaluation of access requests. This is achieved through a vote-based policy evaluation approach. For any given access request all stakeholders with a relevant policy independently evaluate the request w.r.t. to his/her own policy. Once a “allow” or “deny” conclusion is reached the stakeholders vote based on the determination. The votes are then counted and a decision made if the vote exceeds a dynamically produced threshold. Since some stakeholders may consider the requested asset as “very sensitive” while others do not care about it this vote threshold is calculated using the SL from all of the stakeholder’s policies. This is to ensure that stakeholders which consider the asset “very sensitive” have a “bigger say” in the vote.

Similar to RBAC, Friend-centric access control models leverage the already and presumably well maintained lists of Friends. However, this trend appears to conflict with research critical of the privacy setting already employed by leading OSNs based on Friend relationships. As already discussed studies by Lipford et al. [64] and Madejseki et al. [65] show the majority of user’s privacy settings contained errors. Since these settings are already based on schemes where Friends become associated with permissions it is reasonable to theorise other Friend-centric models would experience similarly poor outcomes. Due to the results of Cheek et al. [24] we do no believe this is necessarily the case. In the work [24] the authors are able to demonstrate significant improvement in all areas of a user’s privacy settings by simply presenting Friends in a “better” order.

Besides its use of a novel policy evaluation system , MPAC differentiates itself from the other models so far covered as it allows permission to also be mapped to the names of users and/or groups they are members of. This inclusion of the feature means MPAC can leverage more information readily available in OSNs. Other approaches to OSNs privacy management have adopted context-centric models, such as ABAC which can leverage even more information.

Attribute-Based Access Control (ABAC)

ABAC is a relatively new access control model which has attracted attention in both academia and industry as evident by the recently published set of ABAC guidelines by the National Institute of Standards and Technology (NIST) [51]. ABAC differentiates itself significantly from RBAC and Friend-centric models previously discussed by not relying on a certain information type, such as RBAC being dependent on the existence of roles.

This is achieved by ABAC generalising security relevant characteristics of a user, resource, or the environment to *attributes*. As a result attributes can encompass a wide range characteristics, such as gender, date of birth, school, geographic location, time, etc. Permissions can then be assigned to these attributes and subsequently the entities which hold them. An illustration of this mapping can be found in Figure 1.2. Since a user's OSN profile page will typically be a collection of his/her characteristics ABAC can make use of this information to define privacy policies.

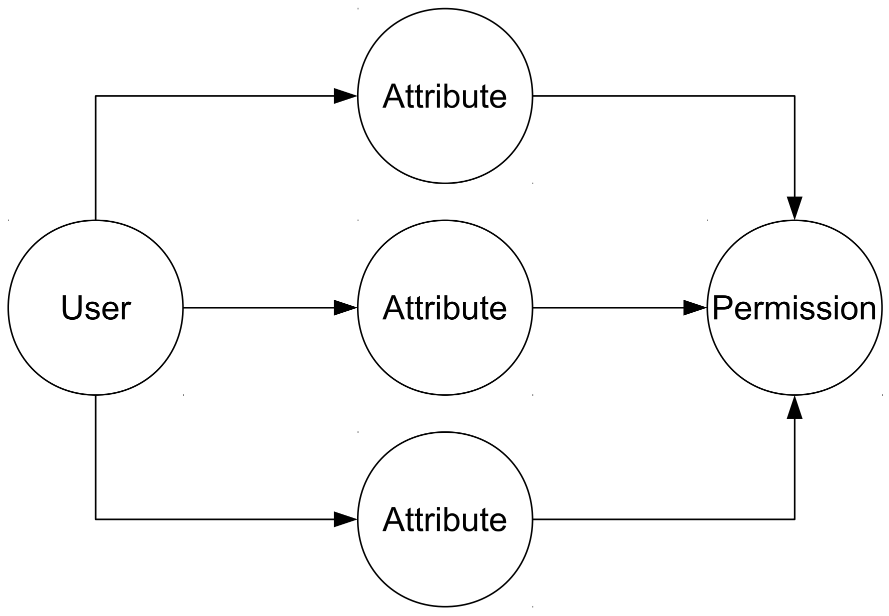


Figure 1.2: ABAC User-Attribute-Permission Mapping

In their work Yuan et al. [86] present an ABAC framework for web-services. This research is of interest for our own as they provide a comprehensive collection of considerations for ABAC systems. Instead of treating attributes as generic characteristics Yuan et al. classify them into three categories; *Subject Attributes*, *Resource Attributes*, and *Environment Attributes*. Subject attributes are characteristics relating to an entity within the system that can take actions. Resource attributes are similar, but relate to entities which are acted upon by subjects, such as a photo or video. Environment attributes represent the current context or state of the environment, such as time of day. By forming policies using combinations of these three attribute types ABAC is capable of fine-grained policies which take into account context. Yuan et al. [86] also note that appropriately named attributes allow ABAC to subsume other models. For instance, Friend attributes effectively subsume Friend-centric models. Yuan et al. assert that RBAC, along with other models, are effective in certain situations, but are not "sufficient to describe the complex, fine-grained access control policies to today's collaborative environments" [86].

Given the newness of the ABAC model it is important that gaps in the literature are identified. Sandhu [74] highlight and outline various issues with ABAC, questioning if this new model is "...a recipe for chaos" [74]. The key advantage of ABAC is its ability to use a wide array of different security relevant information. As seen with the work of Yuan et al. [86] they choose to categorise security relevant information into three different attribute types. This sort of situation concerns Sandhu. When policies are comprised of multiple, independent, possibly conflicting elements how is it possible to maintain predictable and coherent access control results? There are also questions of whether using attributes introduces new attack vectors; such as attribute hiding attacks [28]. Furthermore, in OSNs the use of ABAC presents conflicting goals; the need to form access control policies that protect personal information, but the rules are encoded such that the *very same* information is used to make decisions about protecting it. Sandhu concludes by stating that ABAC needs "...strong conceptual and formal foundations" [74], similar to RBAC96 and the NIST RBAC standard. In the time following Sandhu's comments the NIST published a set of guidelines for ABAC similar to those they produced for RBAC. Though these guidelines are comprehensive they do not address some issues noted by Sandhu. We categorise these issues into three broad areas: *policy expression*, *evaluation*, and *maintenance*.

Important to the design of a ABAC system is how the permission mappings are defined and the semantics that underpin them. Inherent to ABAC is the challenge of expressing the wide array of potentially conflicting sources of security relevant information. As explained by Crampton et al. [28], this often results in ABAC languages being forced to make a trade off between having either a rich set of features or well defined semantics. They [28] tackle this by separating policy target and permission specification into two distinct problems, defining sub-languages for each.

These sub-languages are supported by a consistent set of overlapping semantics defined in ASP, a form of declarative logic programming formally introduced in Section 1.2.3. Informally, an ASP program is an set of inference rules defined over the stable model semantics [42]. These programs are used in conjunction with software tools called *logic engines*, *ASP solvers*, or simply *solvers* to produce conclusions which logically follow from the rules it contains. By defining their semantics in ASP Crampton et al. are able to utilise solvers as the basis of their policy evaluation system [28].

Policy evaluation is the processing of a collection of permission rules in order to determine if a request for access is granted or not. Since policy evaluation is fundamental to any access control model there exists extensive literature on ABAC policy evaluation. In their work Yuan et al. [86] suggest two different approaches for ABAC; one based on First Order Logic (FOL), another based on Extensible Access Control Mark-up Language (XACML).

Yuan et al. [86] suggest that a policy evaluation system for ABAC policies can be based on the evaluation FOL, or variant, expressions. Given ASP is such a variant these comment are consistent with the framework developed by Crampton et al. [28]. Alternatively, Yuan et al. suggest the use of an authorisation structure based on the one used by XACML, a popular XML-based policy language. The position of using XACML is further supported by the NIST ABAC guidelines with it stating XACML is consistent with the ABAC model. The structure of XACML is well known for its support of decentralised systems, indicating ABAC is suitable for decentralised architectures, such as Persona.

When developing an evaluation framework it is important to consider the variety of threats it will be presented with. One of the OSN threats noted by Gao et al. [39] is *Attribute Hiding Attacks*. Since ABAC is particularly vulnerable to these attacks [28] it is important to consider them in the development of our own policy evaluation system. Crampton et al. address the threat of these attacks through the design of their policy evaluation system. Permissions are defined recursively and can be visualised as a tree where each node is an operand, decision or target identifier. Policy targets are evaluated separately to these permissions with the sub-language explicitly supporting the likelihood that target evaluation may not be possible if attributes are missing or withheld. In all of the access frameworks mentioned in this review none of them propose a method for the update of access rules. This is fairly odd as the maintenance of policies is of crucial importance.

Overtime the environment where a set of access permissions and rules operates will change. In the context of OSNs these changes reflect a change in the user's social status, for instance, making new friends, leaving clubs, etc. Kuhn et al. [59] expressing concern over the difficulty of ABAC policy maintenance extends RBAC with attributes to capture the strengths of both RBAC and ABAC. Attributes can be seen as highly dynamic since, according to [59], there is very little up-front effort for setting up ABAC policies, but results in a system that is hard to administer. Whereas the process of determining role structures for RBAC models, known as *role engineering*, is very time consuming, but results in a system that is relatively easy to administer. When texts such as [35] and [83] refer to policy administrators they are clearly referring to some sort of system administrator responsible for policies affecting many users. However, in OSNs who is fulfilling the role of the administrator is less clear.

In OSNs there are effectively two sets of administrators; end users and OSN staff. End users, intuitively, want to maintain their own policies because they wish to retain control over the content they contribute to the network. This makes the end user the administrator of their profile page and associated resources. On the other hand it is still necessary for the OSN operators to be able to apply network wide updates to reflect changes to terms of service or government regulations. We largely ignore the later scenario with our research focusing on the policies maintained by end users and the interactions between different user's policies. As such, we believe the drawback of ABAC, difficulty of policy maintenance, is not as problematic as Kuhn et al. [59] believe. This is because in OSNs the end user is generally responsible for setting and maintaining a single or very small number of policies, not the policies of many. This is also a position supported by the NIST guidelines for ABAC where they assert the relative ease ABAC policy maintenance. Despite this, there does not, to the best of our knowledge, exist any literature on the formal update of OSN privacy settings or ABAC policies. On the other hand, the update of logic programs has been extensively studied [32, 38, 73].

1.2.3 Logic Programming in Access Control

Logic Programming is a programming paradigm based on formal logic. Programs are formed from a collection of logic statements defined over some semantics. These semantics are resolved to derive conclusions using software tools called *solvers* or *logic engines*. These solvers are often of extremely high performance, such as clasp [41], allowing for the fast computation of even very complex logic programs.

In their book, Chin and Older [26] illustrate the strong relationships between formal logic and access control. Using a simple propositional modal logic they express a number of common access control structures and models, such as RBAC. Most importantly they show that formal logic provides a precise mathematical foundation for access control. This combined with the high performance of solvers has resulted in logic programming forming the basis of many access control frameworks [15, 50, 62, 84]

One of these logic programming based frameworks, henceforth shortened to logic-based, is C-Datalog proposed by Bertino et al. [15]. In this research they demonstrate that C-Datalog can express the popular access control models *Bell and La Padula* and RBAC. They also compare their logic-based framework to graph-based access control approaches. Logic-based frameworks implement access control models through the semantics of their underlying logic programs to produce access control models which are logic programs. This provides two key advantages:

1. Policy evaluation systems can be based on the aforementioned logic engines; and,

2. Logic programming techniques can be applied to access control.

C-Datalog [15] is an object oriented derivative of the logic programming language Prolog. C-Datalog allows for the representation of standard access control features and supports the classical object-oriented concepts of classes and inheritance, allowing for the expression of subject and object hierarchies. Since C-Datalog is a logic programming language it is capable of logic programming concepts such as deductive rules, allowing for the expression authorisation and constraint rules. These features result in a highly versatile framework demonstrating logic can accommodate a wide range of features and concepts conducive to access control.

Li et al. [62] present a logical framework for Trust Management (TM) and implement a number of novel features. TM is an approach to access control that makes use of symbolic representations of social trust, for example, a movie ticket authorises the holder to watch the movie at the time and location specified on the ticket, selling or giving the ticket to someone else transfers this authorisation. This ability to "pass on" authorisation has proven useful for decentralised environments. Since OSNs attempt to emulate real world relationships this notion of symbolically representing trust is compelling. Li et al. [62] introduces a new framework based on logic called Delegation Logic (DL). DL implements classical access control features along with the novel features of *delegation* and *thresholds* which are of interest for OSNs.

As its name suggests, DL supports delegation with depth and width restrictions on re-delegation. Delegation can be seen as trusting another principal to handle something on your behalf, while re-delegation is the delegation of an authority you have been delegated. Delegation depth restrictions refers to how "deep" or how many times a particular authority can be re-delegated. Delegation width restrictions controls to whom an authority can be re-delegated to.

The other novel feature, thresholds, are effectively aggregates that only support count and sum operations along with "greater than or equal to" comparisons allowing for the definition of rules such as "Only give loans if 2 or more loan managers approve it". Though Li et al. give no insight as to why they have chosen not to support other common aggregate operations, such as min and max, it appears they do so to avoid conflict with how thresholds are used for delegation depth and width restrictions. Regardless, thresholds highlight the importance and usefulness of aggregates in frameworks for decentralised systems. In the domain of OSNs aggregates would allow for interesting privacy rules such as "Only allow people to view photos with less than 10 comments" or "Only allow people to post to my timeline if we have between 2 and 7 Friends in common". Li et al. [62] also list a number of what they consider necessary features for decentralised access control frameworks, which includes:

- Delegation of attribute authority.
- Attribute-based delegation of attribute authority.
- Inheritance of attributes.
- Conjunction of attributes
- Attributes with fields.

As it can be seen from the texts reviewed there is a significant amount of work on logic-based approaches to access control. Though there are other approaches such as graph-based and reachability-based [30] with the ability to exploit logic programming techniques logic-based approach is compelling for our research. Logic-based approaches are also highly versatile. Logic-based approaches allow for the implementation of popular access control models [15] and can also be used as the foundation for the development entirely new frameworks that include novel features [62] tailored to a specific problem domain.

Answer Set Programming (ASP)

With their seminal paper, Gelfond et al. [42] presented the declarative semantics for logic programs that supports negation, *Stable Model Semantics*, that is the basis for the logic programming paradigm ASP. Emerging in the late 1990s [41] ASP is a form of declarative programming oriented towards complex search problems. It is well suited for declarative knowledge representation and common-sense reasoning, becoming a key technology to imbue software agents with advanced reasoning capabilities. An ASP program is a finite set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_m, \, not\, L_{m+1}, \ldots, not\, L_n. \tag{1.1}$$

Where each L_i is a literal. The above reads as L_0 can safely be assumed to be true if $L_1, \ldots, L_m$ are true and $L_{m+1}, \ldots, L_n$ can safely be assumed to be false. These programs describe a set of knowledge and are used with inference engines, such as DLV [33], to generate *answer sets*. These are sets of conclusions that can be inferred from the knowledge expressed in the program.

Let Π be a program where each rule does not contain default negation literals, and S a set of literals. S is called an answer set of Π, if (i) for each rule, if each literal in the rule body is in S, then the head atom is also in S; and (ii) S is such a minimal consistent set that satisfies condition (i) in terms of set inclusion [11]. If for some atom A, both A and $\neg A$ are in S, then we say that S is the inconsistent answer set of Π, in this case we set S to be the set L of all literals of the language of Π. If S does not contain both A and $\neg A$ then S is consistent, and if every answer set of Π is consistent then Π is also consistent.

Now we consider Π to be a program consisting of rules of the form (1.1), and S be a set of literals. For Π^S, we denote a program obtained from Π as follows:

(i) deleting all rules from Π where for some literal L, *not* L occurs in the body of the rule and $L \in S$; (ii) removing all default negation part of the body from all remaining rules.

Clearly, after such transformation, Π^S will only contain rules without default negation. We say that S is an answer set of Π if it is an answer set of Π^S. For example, the following program $\Pi = \{a \leftarrow \textit{not}\ \neg b.\ c \leftarrow a,\ \textit{not}\ b.\}$ has the answer set $\{a, c\}$.

Possessing properties well suited to access control, such as non-monotonicity and the ability to express domain specific knowledge ASP has gained attention in the computer security research community. Similar to approaches based on traditional logic, ASP based approaches can exploit theoretical results from logic programming, as well as results from ASP research.

As explained by [11] and [19], anything that can be expressed using traditional logic should be representable in ASP. So it should be possible implement all the access control features we have seen so far and those in other logic based approaches in ASP in addition to features that exploit properties of ASP, such as default negation.

Expressing and Reasoning About Policies Using ASP

In Section 1.2.2, the work of Crampton et al. and Hu et al. [28, 50] show the close relation between policy expression and their evaluation. In both of these examples the policy evaluation techniques used are dependent on their schemes having ASP based semantics. Crampton et al. perform model checking to make access control decision using policies defined in ASP. Hu et al. go further, by implementing the MPAC's sophisticated vote-based system as an ASP program.

This relationship between expression and evaluation is further evident by the multitude of logic-based access control frameworks discussed earlier in this section. All of these frameworks [15, 50, 62, 84] use logic programs to express various access control models. This variety in the models highlights the versatility of formal logic as a base for policy expression and reasoning.

With [11] and [19] commenting that ASP can subsume traditional logic programming the research presented in this thesis develops a privacy management framework for OSNs based on ASP. Central to this framework is our own ABAC policy specification language, SocACL, whose semantics are defined as a translation between the language and ASP. Adoption of the ABAC model allows SocACL, and by extension our framework, to support the wide array of security relevant information already present in OSNs by generalising them to attributes. Through SocACL's ASP-based semantics this research also aims to tackle the concerns regarding certain aspects of ABAC raised by Kuhn et al., and Sandhu

[59, 74] previously outlined in Section 1.2.2.

One of the notable strengths of ASP is its ability to express a wide range of domain specific knowledge [11], making it well suited to OSNs. Given Sandhu's concern over ABAC combining a wide range of security information into coherent access control decision the precise and well defined semantics of ASP serves as a suitable semantic base for the model. Our framework's basis on ASP also allows for the novel application of ASP and other logic programming techniques to ABAC and OSNs. Specifically, our framework utilises negotiations for policy evaluation and logic program updates for the update of ABAC policies.

Similar to MPAC's [50] use of ASP to implement weighted voting, our framework employees negotiation based policy evaluation formalised using ASP. Outlined in detail in Chapter 4, this approach reconsiders the typical buyer-seller price negotiation for use in ABAC policy evaluation. In our usage attributes are treated as "currency" by the principals' negotiating access to some resource. Attribute "value" is not encoded using some weighting, but rather employees *attribute disclosure statements* similar to those proposed for Automated Trust Negotiation (ATN) [61]. Attribute disclosure statements, similar to an authorisation rule, define the attribute a principal must posses in order to learn some attribute of another principal.

Policy Update

One of the benefits of the ABAC model, according to the NIST Guidelines [51], is the ease of policy maintenance when compared to other popular models, such as RBAC. As this position conflicts with the earlier concerns voiced by Kuhn et al. [59], who assert ABAC update is hard, it would be reasonable to assume that in the time between the publication of [59] (2010) and the NIST Guidelines (2014) there would be research on the formal update of ABAC policies. However, this is not the case. To the best of our knowledge there is no existing literature on the formal update of ABAC policies. On the other hand the update of logic programs, including ASP, has been extensively studied [32, 38, 73]. This is because, as explained by Eiter et al. [32]:

> ... an agent is situated in an environment which is subject to change. This requests the agent to adapt over time, and to adjust its decision making.

This adjustment to the agent's decision making is done by altering the agent's existing knowledge about its surrounding environment. In other words, as the world changes the agent must update its knowledge so its understanding of the world is more representative of the current world state. Clearly, this holds parallels access control policy update, ABAC or otherwise. Over time the environment which a policy operates within changes, impacting on the correctness of said policy. To ensure that access control decisions resulting from the policy continue to be rational and consistent the policy must be updated w.r.t change in the environment. For example, in the context of OSNs if Alice stops being Friends with Ellen this change has to be reflected in Alice's updated policy otherwise Ellen would still be able to access Alice's resources as a Friend.

In their paper Eiter et al. [32] present a semantic update for ASP programs. Here they reference numerous other papers on the subject of ASP program update, indicating the extensive work in the field. The later sections of the paper are dedicated to comparing their update to others and an analysis of various update properties.

They [32] describe the basis of their update as the *casual rejection principal.* Meaning that rules are rejected only if their is a reason to do so. The reason for rejection adopted by Eiter et al. is remarkably intuitive; new rules always replace old rules. To this end, a rule already held by an agent is replaced if an update request that inherently contains newer, and therefore more correct, rules which contradicts the existing rule. Rules which are not contradictory are simply added to the updated program.

Eiter et al. formally define their update in ASP. Informally, this update does not remove/delete the rejected rules. It simply combines the existing rules with the new rules such that the new rules take precedence when the updated program is solved. Though this approach is effective [32], yielding consistent and intuitive semantics, it results in an updated program which must always become larger over time. Since rules are never deleted an update can only result in the number of rules in the updated program increasing. Additionally, as the program grows over time due to updates its syntax becomes less consistent with the original program. Both of these outcomes can be seen as undesirable, but provide insight into features to consider for our own update.

Firstly, we wish for our ABAC policy update to be syntactical as it allows for easier "at a glance" understanding of the outcome of an update. Second, it should be possible for rules to be removed. As noted by Madejseki et al. [65] when OSN users are presented with an error they are unable to fix it. In the update method developed by Eiter et al. [32] updates are encoded as new rules to include. This approach is problematic in OSNs since it relies on the user being able to determine which rules are causing the unwanted outcomes. For this reason our research considers an update approach where a policy is modified with respect to an observed, yet unwanted, access control outcome, rather than require the user determine *which* rules caused the outcome.

The outcome-focused update holds parallels to the syntactic update presented by Sakama et al. [73] based on abductive reasoning. Sakama et al. present a update framework, formalised in ASP, where an update request consists of two parts; a set of conclusions that are no longer wanted, and a set of new rules to be included in the updated program. Since the details of the this approach are discussed in Chapter 6 we provide a brief explanation of it here. Using ASP Sakama et al. test different combinations of rules from the original program to find ones where the unwanted conclusions no longer hold. These rule combinations are then used to establish which rules are to be deleted from the original program.

Unlike the update of Eiter et al., the approach developed by Sakama et al. meets our three update criteria. By basing their update around of system of a “search then remove” the update meets our first two requirements; syntactic update and rule removal. Including unwanted conclusions in the update request follows well from the conclusion of Madejseki et al. [65]. Assuming users can observe unwanted access control outcomes it seems more reasonable for a user to define an update request in terms of them, rather than having them determine which of their potentially many privacy rules is causing it.

1.3 Thesis Contributions and Outline

Using ASP as the basis of our framework allows the research presented in the thesis to take a novel approach to three aspects of ABAC; policy expression, evaluation, and update. This research has developed a new ABAC policy specification language called SocACL. By defining its semantics as a translation from the language and ASP an equivalence between SocACL and ASP is formed. Using this equivalence we are able to develop a novel framework for policy evaluation based on negotiation. Finally, we address the lack of ABAC policy update literature by leveraging existing ASP program update techniques and adapting them to ABAC. As such the research presented in this thesis has made the following contributions:

- ABAC policy specification language called SocACL.
- ABAC policy evaluation framework based on negotiations.
- ABAC policy update formalism.
- Java implementation prototypes of all of the above.

Chapter 2 formally introduces SocACL, an ABAC policy specification language with semantics defined as a translation between SocACL and ASP. Chapter

4 presents a novel policy evaluation system for ABAC policies based on negotiations to complement SocACL. At some point a SocACL policy would need to be updated to reflect changes to a user's privacy preferences. In Chapter 6 such a ABAC policy update framework is presented.

The formalisms presented in each of these chapters are supported by their respective implementations. In Chapter 3 we present and discuss an implementation of the translation described in Chapter 2. Similarly, in Chapter 5 and Chapter 7 we provide an overview of the implementation of the frameworks presented, respectively, in Chapters 4 and 6. These chapters also detail the performance experiments and results for each prototype, along with the tooling used to perform them.

This thesis concludes in Chapter 8 by summarising all of the previous chapters and providing comments on considerations for the future work.

Chapter 2

Social Access Control Language (SocACL)

2.1 Introduction

This chapter presents SocACL, an ABAC policy specification language for OSNs. We begin with the key concepts behind SocACL, followed by the development of the language since its original publication in [21, 23]. The chapter continues with a comprehensive overview of SocACL's syntax and semantics. This overview also begins to develop the running case study to be used throughout this thesis.

In the following chapter, Chapter 3, we introduce and analyse a Java-based implementation prototype of SocACL called jSocACL.

2.2 Key Concepts

Ideas important to the design of SocACL can be categorised into three areas; OSNs, ABAC and ASP based semantics.

2.2.1 Online Social Networks

The popularity of OSNs is closely tied to many of these networks tailoring their features to specific demographics or fields of interest. For example, Last.FM is tailored around musical tastes, while LinkedIn provides a clean and business-like experience to reflect its focus on professional networks. As a result OSN features often vary between networks. This presents a challenge for the design of SocACL as the language should be able to support a potentially wide range of features.

Besides these demographic focused features OSNs are characterised by the inclusion of social links between users' profile pages and the ability to traverse these links [18]. These links attempt to model real world relationships. As with features, the nature of these relationships can vary between OSNs. For instance, Facebook's Friend relationship models the social or informal concept of a friend, LinkedIn models professional connections, while Twitter models fan followings.

SocACL addresses the above expressive needs by adopting the ABAC model, generalising features and relationships as types of attributes.

2.2.2 Attribute-Based Access Control

Through ABAC SocACL generalises both the OSN specific features and relationships as attributes. However, as noted by Crampton et al. [28] and Sandhu [74] this can be problematic. Crampton et al. assert that ABAC languages are often forced to make a trade-off between functionality and precise semantics. Additionally, Sandhu expresses concern over the ability to produce coherent access control decisions from potentially conflicting sets of attributes.

Crampton et al. [28] address this by defining two sub-languages which work in tandem to define ABAC policies. SocACL addresses these concerns by defining the semantics of SocACL as a translation from SocACL to ASP. Through this translation SocACL inherits the precise and well studied semantics of ASP. This translation also allows for high-performance off-the-shelf ASP solvers to form the basis of SocACL's policy evaluation and maintenance frameworks. We introduce these frameworks in later chapters. The expressive power provided by ASP allows SocACL support the following features:

- Attributes of arbitrary type with fields.
- Direct and indirect relationships of arbitrary type.
- Attribute and relationship inference.
- Positive and negative authorisations with deny override behaviour.
- Aggregates.
- Descriptions.

2.2.3 Answer Set Programming Semantics

As described in Chapter 1, ABAC faces challenges relating to the expression of policies and reasoning about these policies. Despite the ease of ABAC policy maintenance being identified by NIST as a strength of the model [51], there is, to the best of our knowledge, no existing literature on the formal update of ABAC policies.

SocACL tackles these challenges surrounding policy expression, evaluation, and update by exploiting its ASP semantics. The translation creates an equivalence between SocACL Policy Base (PB)s and ASP programs. In Chapters 4 and 6 this equivalence is used to develop novel ABAC policy evaluation and update methodologies based on ASP research.

2.3 SocACL Progression

Since its original publications [21, 23] SocACL has undergone a number of changes. SocACL was initially designed with consideration for the future inclusion of Obligations, described in Section 1.2.2. The feature was never developed beyond the form presented in [21, 23] and has been removed from the version of SocACL presented in this thesis.

SocACL also originally allowed for attributes to specify whether or not they are considered sensitive. Sensitive attributes denote characteristics which a principal may have concerns about sharing, such as their home address, date of birth, etc. This feature was planned to be elaborated upon to support the policy evaluation system presented in Chapter 4. During development sensitivity levels became unnecessary and were then subsequently removed from the language. For historical interest and completeness the old Extended Backus-Naur Form (EBNF) of SocACL containing the syntax of these removed features has been included in Appendix E.

Continued development Obligations was abandoned in favour of improving the query handling framework of SocACL. In [21, 23] query handling is treated as a model checking task with little regard for rule or policy exposure. This has been replaced by the negotiation based query handling framework presented in Chapter 4. Unlike Obligations, SocACL retains the model checking based approach in this chapter to demonstrate how SocACL's ASP based semantics supports "unassisted" policy evaluation.

To support the query approach presented in Chapter 4 SocACL has been also been extended to include *attribute disclosure statements*. Using these statements a principal can define who their attributes are revealed to.

2.4 Policy Base

In SocACL a *Policy Base* (PB) is a finite set of statements which describes the characteristics and access control preferences of a principal. These statements generally take the form (2.1):

$$Prin\ \textbf{says}\ Head\ \textbf{if}\ Body; \tag{2.1}$$

$$Head'_{Prin} \leftarrow Body'. \tag{2.2}$$

Where principal *Prin* is making a statement asserting that *Head* is true if *Body* is true. *Head* can be either an attribute (Section 2.5), direct relationship (Section 2.6.1), authorisation (Section 2.9), or definition (Section 2.6.2 and 2.8) terms to form their respective statements. *Body* is a set of *terms* which forms the statement's decision criteria. This can include attribute, relationship (Section 2.6), constraint, aggregate (Section 2.7), and description (Section 2.8) terms. These terms can be proceeded by the Negation as Failure (NAF) operand "not" in order to use a term's absence as decision criteria, e.g. "not friend". Each statement is terminated by a semicolon (;).

The equations featured throughout this thesis adopt the following conventions. Italicised words denote variables (distinct from SocACL variables), acting as a place holder for an appropriate value. For example, in (2.1) *Prin* could be replaced by the name "alice", while *Head* is replaced by a SocACL term. Regular or bold font characters denote syntax of the formulae. For example, **if** is SocACL syntax.

Semantics of a PB is defined by a transformation to a corresponding ASP program. For a SocACL PB $\mathcal{P}$ its corresponding ASP program is denoted by $Trans(\mathcal{P})$. This is achieved by replacing each SocACL statement of the form (2.1) in $\mathcal{P}$ with the ASP rule (2.2) in the program $Trans(\mathcal{P})$. $Head'_{Prin}$ is a *Head* translated w.r.t. *Prin*, which we clarify in later sections. $Body'$ contains all terms in *Body* translated, also clarified in later sections.

Figure 2.1 shows SocACL expressed in EBNF. A NAME is an atomic value identifying a subject starting with a lowercase letter followed by a sequence of letters, numbers, and underscores, e.g. alice, bob, dennis, paul_12. VARs denote variables, taking the form of a NAME proceeded by a question mark ("?"), e.g. ?car, ?x, ?x2.

SUB is a VAR or NAME which identifies a subject. Similarly, an OBJ is an identifier for an object which can be a VAR or a special instance of NAME where it is enclosed by quotation marks and can contain a full stop, e.g. "cats.jpg", "holiday_movie.mp4".

ACT is an atomic value taking the form of a NAME, denoting some action that can be performed on some object, e.g. read, write, reply. PU is another atomic value, also taking the form of NAME, representing the purpose for this action, e.g. social, commercial, administrative.

```
     Query  =  NAME ‘asks’ NAME · ACT · OBJ · PU‘;’
    Policy  =  {NAME ‘says’ (Rule | Definition) ‘;’}
      Rule  =  Head [‘if’ Body]
Definition  =  Def-RelC | Def-Desc
      Head  =  Auth | Attr  | Rel-Dir
      Body  =  ( BTerm | Aggr | Cons )[‘,’ Body]
     BTerm  =  [‘not’] [Prin ‘says’](Attr | Desc | Rel)
      Auth  =  (‘allow’ | ‘deny’) · Prin · ACT · OBJ · PU
      Attr  =  Prin · ATTR-NAME [ {·Val} ]
  Def-RelC  =  ‘define’ · ‘relchain’ · RCN · ‘(’Body‘)’
  Def-Desc  =  ‘define’ · ‘description’ · DN · VAR · ‘(’Body‘)’
      Aggr  =  VAR ‘=’ Aggr-Op · VAR · ‘(’Body‘)’
            |  Aggr-Op · VAR · ‘(’Body‘)’ · Aggr-Cmp
  Aggr-Cmp  =  (‘exactly’ | ‘atleast’ | ‘atmost’) · Val
            |  ‘between’ · Val · Val
   Aggr-Op  =  ‘count’ | ‘sum’ | ‘min’ | ‘max’
      Desc  =  SUB · ‘description’ · DN
       Rel  =  Rel-Dir | Rel-Sind | Rel-Rind
   Rel-Dir  =  SUB · ‘relationship’ · REL-TYPE · SUB
  Rel-Sind  =  SUB · ‘sindRelationship’ · RCN · SUB
  Rel-Rind  =  SUB · ‘rindRelationship’ · NUM · SUB
      Cons  =  Val (‘<’ | ‘>’ | ‘≤’ | ‘≥’ | ‘=’ | ‘≠’) Val
      Prin  =  SUB | OBJ
       Val  =  NAME | VAR | NUM
```

Figure 2.1: EBNF of SocACL.

ATTR-NAME is simply a NAME for an attribute, e.g. hair_colour, gender, memberOf. While REL-TYPE is VAR or NAME associated with some type of a relationship between subjects, e.g. friend, ?anyRel. DN is a NAME called a *Description Name* used to reference a Description (Section 2.8). RCN is a NAME used to reference a *Relationship Chain* (Section 2.6.2). Finally, NUM is VAR or atomic value denoting an integer value, e.g. ?count, 6.

2.5 Attributes

SocACL attributes are facts about a principal and values associated with it. For instance, Alice has an attribute *eye colour* with the value *brown*. Attributes encompass any security relevant information which does not conform to the other categories outlined in this chapter. To support the wide range of features of different OSNs attributes are of arbitrary type and have any number of fields. They can also be used in two different ways: either as a statement asserting the attribute of some principal or as a term in a statement's *Body*.

2.5.1 Attribute Statements

$$Prin \textbf{ says } P \cdot Attr \cdot Fields \textbf{ if } Body\textbf{;} \tag{2.3}$$

By taking a statement of the form (2.1) and substituting its *Head* with an attribute term one is able to construct *attribute statements* of the form (2.3). These statements read as "$Prin$ asserts principal P", who might be him/herself, holds an attribute named $Attr$ with the associated set of values $Fields$ if all of the terms in $Body$ are true. $Fields = f_1 \cdot ... \cdot f_n$, where f_i is some value associated with this attribute.

The **if** *Body* component of a statement indicates that $Attr$ holds conditionally on the decision criteria in $Body$ being true. In the case there are no decision criteria, such that $Body = \emptyset$, then **if** *Body* can be omitted. Similarly, when $n = 0$ the attribute has no associated values allowing for $\cdot Fields$ to be omitted, for example:

$$\text{alice } \textbf{says} \text{ alice} \cdot \text{married} \tag{2.4}$$

Equation (2.4) contains the stating principal (alice), the asserted holder of the attribute (also alice), and the attribute name (married). The semantics of (2.3) are given by replacing it with the ASP rule (2.5) in the PBs resulting ASP program.

$$Attr(Prin, P, Fields') \leftarrow Body'. \tag{2.5}$$

$Fields' = f_1, ..., f_n$. $Body'$ is the set of atoms in $Body$ which have been translated to their respective ASP equivalent. Though not shown in the above translation, variables used in a SocACL statement are also transformed such that the character immediately after the "?" is replaced by an uppercase letter and the "?" is removed. For example, "?car" becomes "Car" and "?x" is replaced with "X". This is done to accommodate the input language of our ASP solver of choice, DLV, as it considers any axiom that starts with an uppercase letter is a variable.

Example 2.1 (***Alice's Attribute Statements***)

```
alice says alice.married;
alice says alice.hair_colour.brown;
```

```
3 alice says ?Others.enrolled."UoL" if ?Others.memberOf."UoL Tennis";
```

The above example shows three of Alice's attribute statements. Line 1 has Alice stating she holds the attribute married. Line 2 has her stating her hair color is brown. On line 3 Alice infers members of the University of Learning's (UoL) Tennis club, identified by the variable "?Others", are enrolled at UoL. These attributes translate to the following rules:

```
1 married(alice,alice).
2 hair_colour(alice,alice,brown).
3 enrolled(alice,Others,"UoL") :- memberOf(_,Others,"UoL Tennis").
```

In this translation "Others" and "_" are variables while everything else is a NAME, or a derivative of NAME. The variable "_" is called an *anonymous variable* which is described in the next subsection.

2.5.2 Attribute Terms

When attributes are used as decision criteria in a statement's *Body* we call them *attribute terms*. These can take either form (2.6) or (2.7).

$$P \cdot Attr \cdot Fields \tag{2.6}$$

$$Prin \textbf{ says } P \cdot Attr \cdot Fields \tag{2.7}$$

These different forms allow the policy author to define who they trust to assert the attribute. (2.7) requires the attribute *Attr* of principal *P* to be asserted by principal *Prin*, whereas (2.6) states the attribute can be asserted by anybody. This notation provides SocACL with a crude form of delegation as the policy author is able to declare his/her trust in *Prin* to reliably inform them about *P*'s *Attr*. When a statement's *Body* is translated to *Body'*, all of the terms in *Body* are replaced with their corresponding ASP form. Attribute terms of the form (2.6) are replaced by (2.8), while terms of the form (2.7) are replaced by (2.9). For convenience henceforth, when discussing the translation of a SocACL PB translation of a statement's *Body* is always denoted by *Body'*.

$$Attr(\_, P, Fields') \tag{2.8}$$

$$Attr(Prin, P, Fields') \tag{2.9}$$

As it can be seen this translation contains anonymous variables. These denote variables which are never "named" because they are never referenced. Section

3.1.2 provides a more technical explanation of anonymous variables, their implementation, and the technical implications of them. The main difference between (2.8) and (2.9) is the first parameter of each predicate. In (2.9) the first parameter *must* be a reference to a particular principal while in (2.8) the anonymous variable allows this to be substituted with any principals' identifier. Lets begin to introduce a running case study that will be used throughout this thesis. A complete collection of the these PBs can be found in Appendix A.

Example 2.2 (***Running Case Study***)

The scenario describes a principal Alice and her friends in a hypothetical OSN. Alice is an avid photographer and member of a sporting club. She enjoys uploading photos to her profile page and organises them into folders as illustrated in Figure 2.2. In the figure, the folders "public" and "private" are sub-folders of "gallery", and so on. Figure 2.2 can be represented using attributes where an

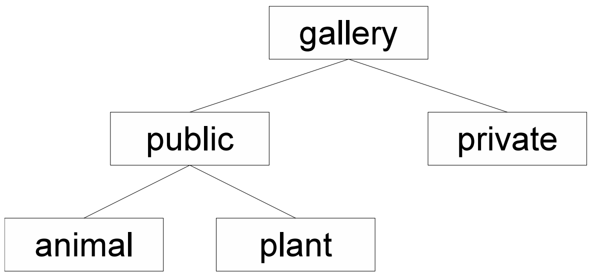


Figure 2.2: Alice's Gallery Folders

object A has an attribute representing which folder it is in.

```
alice says ?A.isIn.public if ?A.isIn.animal;
alice says ?A.isIn.public if ?A.isIn.plant;
```

The above SocACL statements denote the folder `animal` and `plant` are sub-folders of `public`. This translates to:

```
isIn(alice,A,public) :- isIn(_,A,animal).
isIn(alice,A,public) :- inIn(_,A,plant).
```

2.5.3 Attribute Disclosure Statements

As shown later in Chapter 4 there will be scenarios where a principal will want to control how their attributes are revealed. This is done using *attribute disclosure statements*. These statements take on the syntax (2.3) and semantics (2.5) of attribute statements. The key difference is the statement *Body* contains terms related to other principals', while the *Head* contains an attribute of the PB author. For example, consider the following attribute disclosure statement of Alice.

Example 2.3 (***Attribute Disclosure Statement***)

```
1 alice says alice.enrolled."UoL"."Computer Science" if A.enrolled."UoL
    "."Computer Science", not A.memberOf."UoL Robotics";
2 bob says bob.memberOf."UoL Lacrosse" if A.enrolled."UoL".Any;
```

Line 1 has Alice declaring she will reveal that she is enrolled in UoL Computer Science if the someone requesting this attribute is also enrolled in the programme and not also a member of the UoL Robotics club. While on line 2 Bob is willing to reveal he is a member of the UoL Lacrosse club if the requester is enrolled in any UoL course.

2.6 Relationships

Relationships are an integral part of any OSN. SocACL supports three notations for relationships: Direct, Strict-indirect and Relaxed-indirect. For all of these a relationship is a one-directional link between principals' profile pages. For the continuation of this thesis we consider the social graph shown in Figure 2.3. The arrows in this Figure denote the type of relationship each principal believes they are in. For instance, Alice considers Carl a coworker, while to Carl, Alice is a coworker and a friend. Often relationships will be referred to in terms of degrees of separation. This is a widely used term describing the number of "hops" between principals based on their social links [37]. For example, in Figure 2.3 Alice and Bob are in a 1st-degree relationship, while Bob and Ellen's is a 2nd-degree relationship.

2.6.1 Direct Relationships

Direct relationships are 1st-degree links between principals' which may or may not be mirrored. Similar to attributes, direct relationships can either be asserted by a principal using a statement or as part of a *Body* as a term.

Direct Relationship Statements

When a direct relationship is used as the statement *Head* it forms a *direct relationship statement*; (2.10).

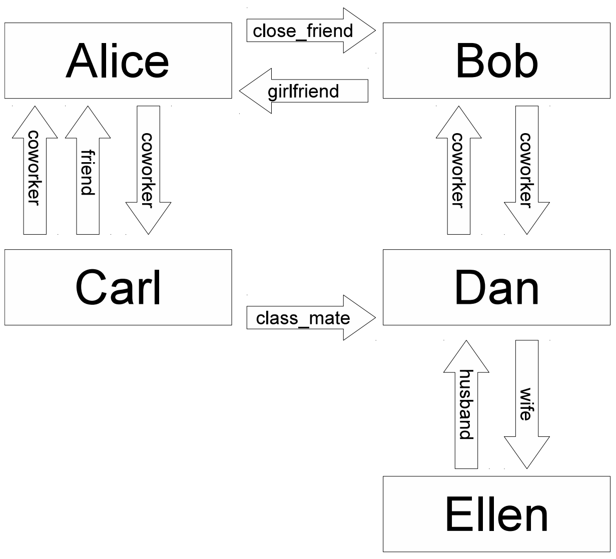


Figure 2.3: A Social Graph.

$$Prin\ \textbf{says}\ P \cdot \textbf{relationship} \cdot rt \cdot Sub\ \textbf{if}\ Body; \tag{2.10}$$

In (2.10) *Prin* is stating that principal *P* believes he/she has a 1st-degree relationship of the type *rt* with principal *Sub*. Translating (2.10) results in the following ASP rule:

$$\textbf{relationship}(Prin, P, Sub, rt) \leftarrow Body', P \neq Sub. \tag{2.11}$$

It can be seen in (2.11) that appended to the translated *Body*, in *Body′* there is an inequality $P \neq Sub$ which forbids the counter-intuitive scenario where a principal is in a 1st-degree relationship with themselves.

Direct Relationship Terms

Similar to attributes, direct relationship terms can take one of two forms in a statement's *Body*. When it is unimportant which principal asserts the relationship during policy evaluation (2.12) is used, while (2.13) is used when it is.

$$P \cdot \textbf{relationship} \cdot rt \cdot Sub \tag{2.12}$$

$$Prin\ \textbf{says}\ P \cdot \textbf{relationship} \cdot rt \cdot Sub \tag{2.13}$$

When a statement's *Body* is translated to *Body′* relationship terms of the form (2.12) are replaced with (2.14) and (2.13) are replaced by (2.15).

$$\textbf{relationship}(\_, P, Sub, rt) \tag{2.14}$$

$$\textbf{relationship}(Prin, P, Sub, rt) \tag{2.15}$$

Example 2.4 (***Direct Relationships***)

Alice and Bob share a rather complicated relationship. Alice considers Bob a `close_friend`, while Bob considers Alice his `girlfriend`:

```
alice says alice.relationship.close_friend.bob;
bob says bob.relationship.girlfriend.alice;
```

Translation of the above would result in:

```
relationship(alice,alice,bob,close_friend) :- alice!=bob.
relationship(bob,bob,alice,girlfriend) :- bob!=alice.
```

In the above the inequalities, `alice!=bob` and `bob!=alice`, are redundant under the *unique name assumption* as they will always be true. As such, they can be omitted.

2.6.2 Indirect Relationships

Indirect relationships define social links between principals of nth-degrees of separation via the relationships of other principals, such as Friend-of-a-Friend. SocACL allows for two approaches to the specification of such relationships; *Strict-indirect* and *Relaxed-indirect*. For both of these relationships evaluation is similar to the node reachability checks of Dhia [30].

Strict-Indirect (Sind) Relationships

Strict-indirect (sind) relationships are indirect links specified as a sequences of direct relationships of a specific type between the two principals. To use sind relationships they must first be defined using a *relationship chain definition statement*, as shown in (2.16).

$$Prin \textbf{ says define} \cdot \textbf{relchain} \cdot RCN \cdot (rt_1, ..., rt_n)\textbf{;} \tag{2.16}$$

$$\begin{aligned}\textbf{sindRelationship}(Prin, P, Sub_n, RCN) \leftarrow \\ \textbf{relationship}(Prin, P, Sub_1, rt_1), \ldots, \\ \textbf{relationship}(Sub_{n-1}, Sub_{n-1}, Sub_n, rt_n), \\ P \neq Sub_1, \ldots, Sub_{n-1} \neq Sub_n. \end{aligned} \tag{2.17}$$

Equation (2.16) reads as *Prin* defines a *Relationship Chain* with the name *RCN* (Relationship Chain Name). This chain is a sequence of direct relationships of the type $rt_i, 1 \leq i \leq n$. Equation (2.17) provides the semantics of (2.16), replacing it in the translated PB. Intuition of these semantics is that for the sind to hold every direct relationship in the definition of *RCN* must hold, be in the correct order and be between unique principals. The requirement of principal uniqueness is to eliminate unexpected policy outcomes resulting from "backtracking" of the relationships.

To reason with sind relationships terms of the form (2.18) are used in a statement's *Body*, with it being replaced by its ASP equivalent (2.19) in *Body'*. Unlike direct relationships, these relationships cannot be proceeded by *Prin* **says** to enforce the source of the relationship information. This is because the semantics of the *RCN* definition (2.17) already have the component direct relationships following the semantics of *Prin* **says**.

$$P \cdot \textbf{sindRelationship} \cdot RCN \cdot Sub \tag{2.18}$$

$$\textbf{sindRelationship}(P, P, Sub, RCN) \tag{2.19}$$

Example 2.5 (*Sind Relationships*)

The following relationship chain definitions are from the social graph shown in Figure 2.3.

```
1 alice says define.relchain.ccm.(coworker, class_mate);
2 alice says define.relchain.ccw.(close_friend, coworker, wife);
3 bob says define.relchain.cocoworker.(coworker,coworker);
```

Line 1 has Alice defining the relationship chain `ccm` which denotes the class mate of a coworker, while line 2 is called `ccw` representing the wife of a coworker of a close friend. On line 3 Bob also defines a relchain where a `cocoworker` is a coworker of a coworker. Translation of these can be seen below:

```
sindRelationship( alice, Sub0, Sub2, ccm ) :- relationship( Sub0, Sub0,
   Sub1, coworker ), relationship( Sub1, Sub1, Sub2, class_mate ), !=(
  Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ).
```

```
sindRelationship( alice, Sub0, Sub3, ccw ) :- relationship( Sub0, Sub0,
    Sub1, close_friend ), relationship( Sub1, Sub1, Sub2, coworker ),
  relationship( Sub2, Sub2, Sub3, wife ), !=( Sub0, Sub1 ), !=( Sub0,
  Sub2 ), !=( Sub0, Sub3 ), !=( Sub1, Sub2 ), !=( Sub1, Sub3 ), !=( Sub2
  , Sub3 ).
sindRelationship( bob, Sub0, Sub2, cocoworker ) :- relationship( Sub0,
  Sub0, Sub1, coworker ), relationship( Sub1, Sub1, Sub2, coworker ),
  !=( Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ).
```

Relaxed-Indirect (Rind) Relationships

Relaxed-indirect relationships specify indirect relationships in terms of distance or degrees of separation (*Depth*) between the principals. Unlike sind relationships, it does not matter what *rt* holds at each degree. This depth is calculated using the ASP rules (2.20), (2.21) and (2.22). These rules derive paths from the direct relationships (2.20), then finds the shortest route between the principals X and Y where each principal along this path is unique using (2.21) and (2.22). These rules are included in every translated SocACL PB as part of the set of rules called *Universal Additions* (UA). Other rules in UA will be introduced as they appear in this chapter. A complete set of UA rules can be found in Appendix A.1

$$\mathbf{path}(X, Y, 1) \leftarrow \mathbf{relationship}(X, X, Y, R, \_). \tag{2.20}$$

$$\begin{aligned}\mathbf{path}(X, Z, D) \leftarrow\ & \mathbf{path}(X, Y, D1), \mathbf{path}(Y, Z, 1), \\ & +(D1, 1, D), X \neq Y, Y \neq Z, X \neq Z.\end{aligned} \tag{2.21}$$

$$\begin{aligned}\mathbf{rindRelationship}(X, X, Y, D) \leftarrow\ & \mathbf{path}(X, Y, \_), \\ & D = \#\mathbf{min}\{D1 : \mathbf{path}(X, Y, D1)\}.\end{aligned} \tag{2.22}$$

To include these relationships in a statement's *Body*, terms of the form (2.23) are used and are replaced by its translation (2.24) in *Body*′.

$$P \cdot \mathbf{rindRelationship} \cdot \mathit{Depth} \cdot \mathit{Sub} \tag{2.23}$$

$$\mathbf{rindRelationship}(P, P, \mathit{Sub}, \mathit{Depth}) \tag{2.24}$$

Example 2.6 (*Rind Relationships*)

Here we have three rindRelationship terms that would be used as part of a statement's *Body*.

```
alice.rindRelationship.dan.2
alice.rindRelationship.ellen.3
bob.rindRelationship.dan.3
```

Considering the social graphs shown in Figure 2.3. Line 1 has Alice asking if Dan is 2-degrees away, while line 2 has her asking if Ellen is 3-degrees away. In both cases, w.r.t. to our social graph this is true. On the other hand, on line 3 Bob asks if Dan is 3-degrees away. This is false as this path cannot be achieved while satisfying the principal uniqueness requirement. Below we find the translation of these terms.

```
rindRelationship(alice,alice,dan,2)
rindRelationship(alice,alice,ellen,3)
rindRelationship(bob,bob,dan,3)
```

2.7 Aggregates

Aggregates encompass the operations count, sum, min, and max. In SocACL these operations are applied over sets of attributes, relationships, or a combination of both. The results of these operations can either be used to form Boolean decisions or be assigned to variables.

2.7.1 Boolean Aggregates

SocACL allows for the results of aggregate operations to be compared against other values to produce a true or false answer. This is done through the inclusion of terms of the form (2.25) in a statement's *Body*.

$$Aggr \cdot (Tar) \cdot (Body) \cdot ACmp \cdot LB \cdot UB \tag{2.25}$$

In (2.25), $Aggr \in \{\textbf{count}, \textbf{sum}, \textbf{min}, \textbf{max}\}$ denotes the operation to be applied to Tar in *Body*. The *Body* of an aggregate can contain a combination of attributes and relationships which define the conditions where the variable Tar holds some value for which the operation is performed. In other words, *Body* describes the Tar.

- **count** provides the cardinality of the set of all instances of Tar which hold in *Body* for example, a count of the Friends which two principals' have in common.
- **sum** results in the total value of Tar, such as total of number of "likes" received by all of Alice's holiday photos.
- **min** is the smallest value of Tar w.r.t. *Body*, the photo with the fewest "likes".
- **max** is the largest value of Tar w.r.t. *Body*, the photo with the most "likes".

The result of the aggregate is compared to the integer values LB and UB, respectively representing the lower and upper bound, based on $ACmp \in \{$**exactly**, **atleast**, **atmost**, **between**$\}$. Note that UB is only used for **between** and can otherwise be omitted.

$$L\ \#Aggr\{Tar : Body'\}\ U, Body'_{Tar} \tag{2.26}$$

The semantics of (2.25) is defined by a transformation to (2.26). L and B are substituted depending on $ACmp$. These substitutions are summarised in Table 2.1.

$ACmp$	L	U
exactly	-	$= LB$
atleast	$LB \leq$	-
atmost	-	$\leq LB$
between	$LB \leq$	$\leq UB$

Table 2.1: $ACmp$ translation substitutions.

$Body'_{Tar}$ is a translation of *Body* in the aggregate where instances of Tar are replaced with Tar', a variant of Tar where $Tar \neq Tar'$. This accounts for the ASP solver DLV. DLV's implementation of aggregates, on which SocACL's semantics are based, does not allow for Tar to occur outside of the curly braces.

Example 2.7 (***Boolean Aggregates***)

The below holds true if $Sub \geq 2$ where Sub is the number of direct relationships of any type which Alice and Bob have in common.

```
count.(?Sub).(alice.relationship.?Any.?Sub,bob.relationship.?Any.?Sub).
  atleast.2
```

Replacing `?Any` with “_” yields to following translation:

```
2<=#count{Sub:relationship(_,alice,Sub,_),relationship(_,bob,Sub,_)},
  relationship(_,alice,Sub1,_),relationship(_,bob,Sub1,_)
```

2.7.2 Assignment Aggregates

Alternatively, the results of an aggregate can be assigned to a variable to include values calculated at runtime in a statement. This is done through the inclusion of (2.27) in a statement's *Body* and its translation (2.28) in *Body'*. In both V denotes some variable to which the result is assigned and all other variables are consistent with their previous definitions.

$$V = Aggr \cdot (Tar) \cdot (Body) \tag{2.27}$$

$$V = \#Aggr\{Tar : Body'\}, Body'_{Tar} \tag{2.28}$$

Example 2.8 (***Assignment Aggregates***)

Alice prepares a dynamically calculated attribute statement which provides a count of all of her direct relationships.

```
alice says alice.friendCount.?A if ?A = count.(?Sub).(alice says alice.
  relationship.?Any.?Sub);
alice says alice.mostPopular.photo.?Object if ?Object.type.photo, ?
  Object.likes.?V, ?V=max.(?L).(?Obj.likes.?L, ?Obj.description.
  animalPhoto);
```

Which translates to:

```
friendCount(alice,alice,A) :- A=#count{Sub:relationship(alice,alice,Sub
  ,_)}.
mostPopular(alice,alice,photo,Object) :- type(_,Object,photo), likes(_,
  Object,V), V=#max{L:Obj.type.photo, Obj.likes.L}.
```

2.8 Descriptions

Descriptions allow the policy specifier to group decision criteria to form a description of a principal.

2.8.1 Defining a Description

Similarly to sind relationships 2.6.2, descriptions must be defined before their terms can be used in a statement's *Body*. This is done through using *description definition* statements which take the form (2.29).

$$Prin \textbf{ says define} \cdot \textbf{description} \cdot DN \cdot P \cdot \textbf{(}Body\textbf{);} \tag{2.29}$$

Where DN is the *description name* used to reference the description of principal P given by *Body*. As shown by its translation (2.30) that a description is simply mapping the decision criteria in *Body* to another predicate. In turn, this predicate can be used in a statement specification instead of the *Body*. This translation replaces (2.29) in the translated SocACL PB.

$$\textbf{description}(Prin, P, DN) \leftarrow Body'. \tag{2.30}$$

Example 2.9 **(*Description Definition Statement*)**

Alice decides to define a new description to make it easier to specify policies over her ever growing collection of plant photos. We see below that object Object fits the description of `plantPhoto` if it is a photo in the `plant` folder. Line 2 shows its translation.

```
1 alice says define.description.plantPhoto.?Object.(?Object.isIn.plant, ?
    Object.type.photo);
2 description(alice,Object,plantPhoto) :- isIn(_,Object,animal), type(_,
    Object,photo).
```

2.8.2 Description Terms

Descriptions are used in a similar fashion to attributes by including (2.31) in a statement's *Body*, with it being replaced by its translation (2.32) in *Body*′. (2.31) is read as "P fits the description DN" and *Prin* is always the principal who authored the policy.

$$P \cdot \mathbf{description} \cdot DN \tag{2.31}$$

$$\mathbf{description}(Prin, P, DN) \tag{2.32}$$

Example 2.10 (***Description Terms***)

Alice infers a new attribute from the description of a plant photo on line 1. Line 2 shows its translation

```
alice says ?Object.photoOf.plants if ?Object.description.plantPhoto;
photoOf(alice,Object,plants) :- description(alice,Object,plantPhoto).
```

2.9 Authorisations

SocACL supports both positive and negative authorisations with deny override behaviour. An authorisation can only be used to replace the *Head* of a statement to form an *authorisation statement* as follows:

$$Prin\ \mathbf{says}\ Perm \cdot P \cdot Act \cdot Obj \cdot Pu\ \mathbf{if}\ Body\mathbf{;} \tag{2.33}$$

$Perm \in \{\mathbf{allow}, \mathbf{deny}\}$ is the permission type, allowing or denying a principal P from performing action Act on object or resource Obj for the purpose Pu. $Body$ specifies the conditions under which the authorisation holds. The $Body$ can also be used to dynamically specify P or Obj to which this statement applies. Purpose Pu restricts the permission to a specific purpose. For example, Bob is comfortable with allowing anyone to access his photos for "social" purposes, but denies access for "commercial".

$$Perm(Prin, P, Act, Obj, Pu) \leftarrow Body' \tag{2.34}$$

$$\begin{aligned}&\mathbf{action(}P, Prin, Act, Obj, Pu\mathbf{)} \leftarrow \\ &\qquad \mathbf{allow(}Prin, P, Act, Obj, Pu\mathbf{)}, \\ &\qquad \mathbf{not\ deny(}Prin, P, Act, Obj, Pu\mathbf{).}\end{aligned} \tag{2.35}$$

Queries make a request to perform a specific action on an object, rather than a direct request for the authorisation. This is done through the ASP rule (2.35) which is included in every translated SocACL PB and is part of the UA. These queries are, generally, successful (granting permission to perform the action) if there a positive authorisation holds and there is no negative authorisation which overrides it. Two different approaches to query handling are outlined in detail in the following section (Section 2.10) and in Chapter 4.

Example 2.11 (***Authorisation Statement***)

Alice wishes to allow access to Objects for social purposes if the accessor is within 2 degrees of separation and the Object fits the description of "plantPhoto".

```
alice says allow.?Other.view.?Object.social if alice.rindRelationship.?
  A.?Other, ?A <= 2, ?Object.description.plantPhoto;
```

Translation below:

```
allow(alice,Other,view,Object,social) :- rindRelationship(alice,alice,
  Other,A), A <= 2, description(alice,Object,plantPhoto).
```

2.10 Basic Queries

A query is a request from one principal $Prin$ to another, P, asking to perform action Act on object Obj for the purpose Pu. Queries in SocACL can either follow the model checking based approach proposed in [21] and outlined in this section or the negotiation based model proposed in [22] and presented in Chapter 4. Regardless of the approach all SocACL queries to a PB take the form (2.36), whose ASP translation is as described in the form (2.37).

$$Prin\ \textbf{asks}\ P \cdot Act \cdot Obj \cdot Pu; \tag{2.36}$$

$$\textbf{action}(Prin, P, Act, Obj, Pu) \tag{2.37}$$

Query answering is an assignment of truth values to a query w.r.t. a PB. Using the model checking approach, this is done by computing the models of the ASP translation of a PB w.r.t. the translated query. These computations are performed using an ASP solver, in our case, DLV [60].

Let $\mathcal{P}$ be a SocACL PB and ϕ be a query of the form (2.36). For $\mathcal{P}$ there is the ASP program $Trans(\mathcal{P})$ produced by translating every SocACL statement of the form (2.1) in $\mathcal{P}$ to a ASP rule (2.2) as per the methodology outlined in this chapter. Similarly, the translation of query ϕ is denoted by $trans(\phi)$, where $trans(\phi)$ is the transformation ϕ of the form (2.36) to (2.37).$\mathcal{P} \models \phi$ iff $Trans(\mathcal{P}) \models trans(\phi)$. Meaning that $\mathcal{P}$ satisfies query ϕ, answers yes, if and only if $trans(\phi)$ is satisfied in every answer set of $Trans(\mathcal{P})$, where $trans(\phi)$ is the translation of ϕ and program $Trans(\mathcal{P})$ is the translation of $\mathcal{P}$

Programs resulting from the translation are *Normal Logic Programs* (NLPs) with arbitrary nonmonotonic negation, and may include aggregates. Since the complexity results of reasoning over these types of programs are already known we reference the results of [33] who show that the complexity of this problem is co-NP-complete when the program does not contain aggregates, and raises to Π_2^P-complete when it does.

Example 2.12 **(*Basic Queries*)**

This example considers the PB of Alice which contains all of the SocACL statements from the previous examples in addition to attributes for the photos `cats.jpg` and `dogs.jpg`, both of which are in the `animal` folder. These attributes can be found as part of Alice's PB in Appendix A. Running DLV with the "-filter=action" option results in the output of all actions that can be performed as a result of the authorisations in the translation of Alice's PB.

```
{action(bob,alice,view,"cats.jpg",social), action(bob,alice,view,"dogs.
  jpg",social),action(carl,alice,view,"cats.jpg",social), action(carl,
  alice,view,"dogs.jpg",social), action(dan,alice,view,"cats.jpg",social
  ), action(dan,alice,view,"dogs.jpg",social)}
```

In the above answer set Bob, Carl, and Dan can view both photos for social purposes. Ellen on the other hand cannot since she is not in at least a 2nd-degree relationship with Alice. To further demonstrate this we apply two queries to the policy, one for Carl and another for Ellen asking for permission to view "cats.jpg" for social purposes. Below we show the queries from both Carl and Ellen immediately followed by their respective translation.

```
carl asks alice.view."cats.jpg".social;
action(carl,alice,view,"cats.jpg",social)

ellen asks alice.view."cats.jpg".social;
action(ellen,alice,view,"cats.jpg",social)
```

Applying both queries to the translated PB using DLV's built in query system yields the below output. Ellen cannot view the photo while Carl can. These results are based on the answer sets DLV was able to generate based on Alice's

PB. For Ellen's query DLV concludes it is false since not all of the answer sets of Alice's policy based can satisfy the query. Whereas, Carl's query can be satisfied by them, and is therefore true.

```
action(carl,alice,view,"cats.jpg",social) is cautiously true.
action(ellen,alice,view,"cats.jpg",social) is cautiously false.
```

2.11 Chapter Summary

In this chapter we have introduced and discussed the ABAC language, SocACL. In Section 2.2 we outlined the concepts key to the design of the language. Following this, in Section 2.3 we noted the changes made to SocACL since its original publication in [21, 23].

With these introductory sections complete we presented the syntax and semantics of SocACL. This began with a formal outline of SocACL PBs in Section 2.4. In Section 2.5 we outlined attributes, while Section 2.6 introduced SoACL's various relationship constructs. SocACL aggregates were presented in Section 2.7 and in Section 2.8 we outlined descriptions. Finally, authorisation statements were introduced in Section 2.9.

The chapter concluded with Section 2.10 where we provided a model checking based approach to SocACL query handling.

Chapter 3

Implementation of SocACL

3.1 Introduction

In this chapter we introduce jSocACL, a Java implementation of the SocACL to ASP translation outlined in Chapter 2.

3.1.1 System Overview and Technical Details

jSocACL is constructed around a parser/lexical analyser generated from a ANTLR3 grammar [71]. ANTLR3 was selected in favour of immensely popular flex/bison suite because it provided Java as an output language. jSocACL was originally developed under the assumption that SocACL policies would be evaluated using the model checking approach outlined in Section 2.10. As such, jSocACL integrates with DLV using DLVWrapper v4.2, a collection of Java interfaces for the popular ASP solver. Figure 3.1 summarises jSocACL's organisation.

jSocACL takes a SocACL PB as the sole input or in combination with a SocACL query for Basic Queries (Section 2.10). Both the PB and query are translated using the SocACL ANTLR3 compiler to get ASP representations of each.

The translated query is used with DLV (shown as ASP Solver in Figure 3.1) while the PB undergoes post-processing, a process not previously introduced. Since the reasoning behind the need for the post-process is rather complex the following section is dedicated to this, Section 3.1.2.

Once the post-processing is complete the PB is stored. The PB is then either used with the ASP solver to perform Basic Queries or directly output from jSocACL.

When used as part of a larger system, jSocACL does not directly interact with DLV. In this scenario jSocACL simply performs the SocACL to ASP translations to provide input for the implementations of the frameworks described in Chapters

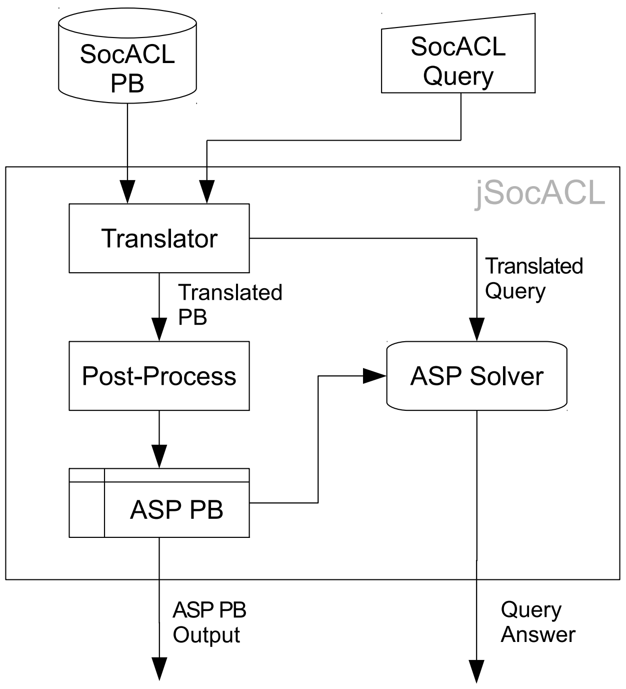


Figure 3.1: jSocACL Flow Chart.

4 and 6.

3.1.2 Considerations for Post-Processing

The development of jSocACL highlighted a number of technical considerations. Since these are largely unrelated to the translation, but rather to our use of off-the-shelf ASP solvers, we consider them separately in this subsection. The solutions to the technical issues outlined here are implemented in jSocACL by the Post-Process module shown in Figure 3.1.

DLV Rule Safety

As mentioned in Section 2.7 the SocACL to ASP translation takes into account the syntactical quirks of DLV. However, DLV imposes additional requirements on input programs called *rule safety.* Rule safety refers to a set of rules placed over variables used in ASP programs intended as input for DLV. As explained in the DLV user manual [79]:

> DLV imposes a safety condition on variables in rules. This guarantees that a rule is logically equivalent to the set of its Herbrand instances.

In other words, rule safety describes the conditions where DLV can ensure all variables contained within a program's rules can be grounded. The manual continues by defining variable and rule safety.

Definition 3.1 (***DLV Variable and Rule Safety***)

A variable x in an aggregate-free rule is safe if at least one of the following conditions is satisfied:

1. x occurs in a positive standard predicate in the body of the rule.

2. x occurs in a true negated standard predicate in the body of the rule.

3. x occurs in the last argument of an arithmetic predicate A and all other arguments of A are safe.

A rule is safe if all of its variables are safe. However, cyclic dependencies are disallowed.

Lets consider these conditions w.r.t. SocACL's variable use. Condition 1 requires that a SocACL statements which have variables in its *Head* must also have those variables in its *Body*. This condition is easily satisfied by careful policy authoring. Condition 3 relates to the use of DLV's inbuilt arithmetic predicates and is satisfied by SocACL's variable constraints use of infix notation. *Cyclic dependencies* relate to the use of DLV built-ins, for example:

```
a(Z) :- node(X), #count{V : edge(V,Z)}=Y, Z=X+Y.
```

The above ASP rule contains a cyclic dependency. This is because the `#count` aggregate produces a result used by the addition `Z=X+Y`, but the aggregate needs the result of `Z=X+Y` before it can produce this result. As with condition 1, cyclic dependencies can be avoided through careful policy authoring. This leaves Condition 2 for special consideration. Lets begin by considering the following SocACL statement followed by its translation:

```
1 alice says allow.?A.view."cats.jpg".social if ?A.memberOf."UoL Lacrosse
  ", not ?A.memberOf."UoL Tennis";
2 allow( alice, A, view, "cats.jpg", social ) :- memberOf( _, A, "UoL
  Lacrosse" ), not memberOf( _, A, "UoL Tennis" ).
```

This statement contains a Separation of Duty (SoD) condition where principal A, in order to gain access to the picture, cannot be a member of both clubs. The statement's translation found on line 2 is *not* safe, because the variables it contains are not safe as per Definition 3.1. Variable A is safe because it occurs in both a positive and a negative predicate, `memberOf` and `not memberOf`, respectively in the rule body. On the other hand, "_" is not safe despite occurring in the same predicates as A. Though this result appears counterintuitive it makes sense when how DLV implements the anonymous variable "_" is understood. Internally DLV, as explained by the DLV Manual [79], replaces every instance of variable "_" in a rule with a uniquely named variable, such that within DLV the example translation is actually the following rule:

```
allow( alice, A, view, "cats.jpg", social ) :- memberOf( V1, A, "UoL
  Lacrosse" ), not memberOf( V2, A, "UoL Tennis" ), !=( A, alice ).
```

Replacing "_" with the variables V1 and V2 (DLV does not actually use *these* variables, the names were picked for the sake of the example). Clearly after this replacement, V1 is safe while V2 is not. Since it is possible to make these rules safe without user involvement, jSocACL does so by performing a post-process on translated PBs.

First, predicates of the same name, in our example they are both `memberOf`, are grouped if they denote a SoD requirement. In these groups variable "_" in the same parameter positions are replaced with named variables specific to the group. For instance our example rules becomes:

```
allow( alice, A, view, "cats.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Tennis" ), !=( A,
  alice ).
```

The above rule contains a group of `memberOf` predicates which represent a SoD requirement. The first parameter of each predicate in the group is replaced with the named variable SODFIX_0 (SoD fix zero). Though the anonymous variables now have names they are never referenced outside of the group. Since this technique does not burden the policy author with naming variables it does not interfere with the intended purpose of anonymous variables. However, it should be noted that in this example there is a degree of semantic loss as the first parameter of the predicate is used to specify the source of these attributes. Originally the rule stated any principal could be the source either attribute, now the rule requires both are sources from the same principal. Condition 2 also causes problems with rules such as:

```
1 alice says alice.enrolled."UoL"."Computer Science" if not ?A.memberOf."
    UoL Robotics";
2 enrolled( alice, alice, "UoL", "Computer Science" ) :- not memberOf( _,
     A,"UoL Robotics" ).
```

The above is an example of an attribute disclosure statement. This statement does not contain a SoD requirement, but its translation, line 2, has a "_" being used in a NAF predicate which makes the rule unsafe. This scenario is again approached by post-processing the translated PB.

For these types of rules jSocACL introduces “helper” predicates and rules to “move” the anonymous variable aways from the NAF operand. Doing this replaces our translated example with the following rules:

```
1 enrolled( alice, alice, "UoL", "Computer Science" ) :- not hh_memberOf(
    A, "UoL Robotics" );
2 hh_memberOf( HH0, HH1 ) :- memberOf( _, HH0, HH1 ).
```

As it can be seen the problematic predicate in the unsafe rule has been replaced with the helper predicate; `hh_memberOf`. The helper is also used to form a new rule on line 2. In this new rule the problematic predicate forms the new rule’s Body, but with the NAF operand removed. Semantically this construct is identical to the original rule, but is considered safe under DLV’s requirements.

3.2 Translation Algorithms

Each of these algorithms is based on the SocACL to ASP formalisms presented throughout this chapter. Unlike the following chapters, the implementation of jSocACL lacks a detailed overview of the Java classes which form the prototype. This is due to jSocACL being largely dependent on “black box” code generated by ANTLR3 w.r.t. our defined SocACL to ASP grammar. Since the Java code generated by ANTLR3 is, by design, poorly human readable, this section focuses on the algorithms on which the grammar was based.

3.2.1 TransPB

Algorithm 1, TransPB, takes a SocACL PB as input and translates it to yield the ASP program ASPOut. Each Statement in SocACLPB is tested to see what type of statement it is. Depending on its type the statement is translated by the appropriate function and the function’s result being appended to ASPOut. The set of rules, UA, known as the Universal Additional, are included in the final output by appending them to ASPOut.

3.2.2 TransAttribute

Algorithm 2, TransAttribute, takes an attribute term as input and then tokenises it, storing the tokenised version in the array t_tok. These tokens are then assigned names which match those used in Section 2.5. If the attribute contains fields the values are processed by TransFields to normalise nuances for particular ASP solvers. The TransFields algorithm is not provided and is left as an application specific consideration.

Algorithm 1: TransPB

```
   Input  : SocACLPB
1  Let UA be set of ASP rules called the Universal Additions;
2  foreach Statement in SocACLPB do
3      switch Statement is a... do
4          case Attribute
5              ASPOut = ASPOut + TransAttrStatement(Statement);
6          case Relationship
7              ASPOut = ASPOut + TransRelStatement(Statement);
8          case Authorisation
9              ASPOut = ASPOut + TransAuthorisation(Statement);
10         case Description Definition
11             ASPOut = ASPOut + TransDefDesc(Statement);
12         case Relchain Definition
13             ASPOut = ASPOut + TransRelchain(Statement);
14     ASPOut = ASPOut + UA;
       Output: ASPOut
```

Algorithm 2: TransAttribute

```
   Input  : attrTerm
1  t_tok = Tokenise( attrTerm );
2  if t_tok[1] == "says" then
3      Prin = t_tok[0];
4  else
5      Prin = "_";
6  P = t_tok[2] ;
7  attr = t_tok[3] ;
8  fields = TransFields( t_tok[4] );
9  ASPOut = attr + "(" + Prin + "," P + fields + ")";
   Output: ASPOut
```

3.2.3 TransRelationship

Algorithm 3, TransRelationship, takes a direct, sind or rind relationship term and translates it to its ASP equivalent predicate. First the term is tokenised by Tokenise() with the result being stored in the array t_tok. These tokens are again assigned names which align with the formalism presented in Section 2.6. After which, the tokens are rearranged to form the ASP predicate ASPOut.

3.2.4 TransDescription

Algorithm 4, TransDescriptions, takes a description term and translates it to its equivalent ASP predicate. Again, the input is tokenised using Tokenise() and the

Algorithm 3: TransRelationship

```
Input  : relTerm
1  t_tok = Tokenise(relTerm);
2  if t_tok[1] == "says" then
3      Prin = t_tok[0];
4  else
5      Prin = "_";
6  P = t_tok[2];
7  relCat = t_tok[3];
8  rt = t_tok[4];
9  Sub = t_tok[5];
10 ASPOut = relCat + "(" + Prin + "," + P + "," + Sub + "," + rt + ")";
Output: ASPOut
```

result stored in the array t_tok. After assigning similar names to Section 2.8 the tokens are rearranged to form the ASP predicate ASPOut.

Algorithm 4: TransDescription

```
Input  : descTerm
1  Let Prin be the identifier of the policy author.
2  t_tok = Tokenise(descTerm);
3  P = t_tok[0];
4  DN = t_tok[2];
5  ASPOut = "description(" + Prin + "," + P + "," + DN + ")";
Output: ASPOut
```

3.2.5 TransAggregate

Algorithm 5, TransAggregate, takes a aggregate term and translates it to its equivalent ASP predicate. After the term has been tokenised to form the array t_tok it is determined if it is either a assignment or a Boolean aggregate. From there the tokens are rearranged to form either ASPOut for assignment aggregates, or a AggreCore for Boolean aggregates. For Boolean aggregates, once the AggrCore is formed the upper and lower bounds are attached to it depending on the value of ACmp to yield ASPOut.

3.2.6 TransDelRelchain

Algorithm 6, TransRelchain, takes a relationship chain definition statement and prepares an ASP rule from it. Once the statement has been tokenised the set of relationship types, RT, is processed. For each relationships type, rt, in RT

Algorithm 5: TransAggregate

```
Input : aggrTerm
t_tok = Tokenise(aggrTerm);
if t_tok[1] == "=" then
    V = t_tok[0];
    Aggr = t_tok[2];
    Tar = t_tok[3];
    TBody = TransBody( t_tok[4] );
    TBodyTar = Replace(Tar, TBody, Tar+"1" );
    ASPOut = V + "=#" + Aggr + "{" + Tar + ":" TBody + "}," +
    TBodyTar;
else
    Aggr = t_tok[0];
    Tar = t_tok[1];
    TBody = TransBody( t_tok[2] );
    TBodyTar = Replace(Tar, TBody, Tar+"1" );
    ACmp = t_tok[3];
    AggrCore = "#" + Aggr + "{" + Tar + ":" TBody + "}," +
    TBodyTar;
    switch ACmp == do
        LB = t_tok[4];
        case "exactly"
            ASPOut = AggrCore + "=" + LB;
        case "atleast"
            ASPOut = LB + "≤" + AggrCore;
        case "atmost"
            ASPOut = AggrCore + "≤" + LB;
        case "between"
            UB = t_tok[5];
            ASPOut = LB + "≤" + AggrCore + "≤" + UB;
Output: ASPOut
```

there is a relationship predicate which forms part of the ASPBody. Tokens are rearrange to also form the ASPHead to which ASPBody is appended.

3.2.7 TransBody

Algorithm 7, TransBody, processes a conjunction of SocACL terms, known as a Body, to form a conjunction of ASP predicates. The body is tokenised such that each token is a SocACL term. These terms are iterated over and the type of term it is checked so it can be translated accordingly.

Algorithm 6: TransDefRelchan

```
Input   : RelchainStatement
s_tok = Tokenise(RelchainStatement);
n = 1;
Prin = s_tok[0];
RCN = s_tok[4];
RT = s_tok[5];
foreach rt ∈ RT do
    SubN = Sub + (n-1);
    SubNNext = Sub + n;
    DirRelN = “relationship(” + Prin + “,” + SubN + “,” + SubNext +
    “,” + rt + “),” + SubN + “≠” + SubNext;
    if not last rt ∈ RT then
        DirRelN = DirRelN + “,”;
    ASPBody = ASPBody + DirRelN;
    n++;
ASPHead = “sindRelationship(” + Prin + “,” + Sub + n + “,” + RCN +
“)”;
ASPOut = ASPHead + “:-” + ASPBody + “.”;
Output: ASPOut
```

Algorithm 7: TransBody

```
Input   : Body
b_tok = Tokenise(Body);
foreach term ∈ b_tok do
    switch term is a... do
        case attribute
            ASPOut = ASPOut + TransAttribute(term);
        case relationship
            ASPOut = ASPOut + TranRelationship(term);
        case aggregate
            ASPOut = ASPOut + TransAggregate(term);
        case description
            ASPOut = ASPOut + TransDescription(term);
        case constraint
            ASPOut = ASPOut + term;
    if not last term ∈ b_tok then
        ASPOut = ASPOut + “,”;
Output: ASPOut
```

3.2.8 TransAuthorisation

Algorithm 8, TransAuthorisation, translates SocACL authorisation statements to an ASP rule. After tokenising the statement the ASPAuthOut is prepared, with any Body terms attached to the ASPAuthOut if needed.

Algorithm 8: TransAuthorisation

```
   Input  : AuthStatement
 1 s_tok = Tokenise(AuthStatement);
 2 Prin = s_tok[0];
 3 Perm = s_tok[2];
 4 P = s_tok[3];
 5 Act = s_tok[4];
 6 Obj = s_tok[5];
 7 Pu = s_tok[6];
 8 ASPAuthHead = Perm + “(” + Prin + “,” + P + “,” + Act + “,” + Obj
   + “,” + Pu + “)”;
 9 if s_tok[7] then
10     Body = s_tok[8];
11     ASPAuthBody = TransBody(Body);
12     ASPAuthOut = ASPAuthHead + “:-” + ASPAuthBody + “.”;
13 else
14     ASPAuthOut = ASPAuthHead + “.”;
15 ASPOut = ASPAuthOut;
   Output: ASPOut
```

3.2.9 TransRelStatement

Algorithm 9, TransRelStatement, translates SocACL relationship statements to their ASP equivalent rule. Once the statement has been tokenised these tokens are arranged to form the ASPHead. If the statement has a Body, then it is translated and appended to ASPHead to form ASPOut.

Algorithm 9: TransRelStatement

```
   Input  : RelStatement
 1 s_tok = Tokenise(RelStatement);
 2 Prin = s_tok[0];
 3 P = s_tok[2];
 4 rt = s_tok[4];
 5 Sub = s_tok[5];
 6 ASPHead = “relationship(”+ Prin + “,” + P + “,” + Sub + “,” + rt +
   “)”;
 7 if s_tok[6] == “if ” then
 8     ASPBody = TransBody(s_tok[6]);
 9 ASPBody = P + “$\neq$” + Sub + “,” + ASPBody;
10 ASPOut = ASPHead + “:-” + ASPBody + “.”;
   Output: ASPOut
```

3.2.10 TransAttrStatement

Algorithm 10, TransAttrStatement, translates attribute statements to ASP rules. After tokenising the statement it is determined if the statement either;

- Has no fields nor Body; or
- Has fields, but no Body; or
- Has no fields, but has Body; or
- Has fields and a Body.

Depending on this check the fields and Body are translated appropriately and appended to the ASPHeadBase to form ASPOut.

Algorithm 10: TransAttrStatement

```
   Input   : AttrStatement
1  s_tok = Tokenise(AttrStatement);
2  Prin = s_tok[0];
3  P = s_tok[2];
4  Attr = s_tok[3];
5  ASPHeadBase = Attr + “(” + Prin + “,” + P;
6  if s_tok[4] == “;” then
7      ASPHead = ASPHeadBase + “).”;
       Output: ASPHead
8  else if s_tok[5] == “;” then
9      fields = TransFields(s_tok[4]);
10     ASPHead = ASPHeadBase + “,” + fields + “).”;
       Output: ASPHead
11 else if s_tok[4] == “if” then
12     Body = s_tok[5];
13     ASPBody = TransBody(Body);
14     ASPHead = ASPHeadBase + “)”;
15     ASPOutput = ASPHead + “:-” + ASPBody + “.”;
       Output: ASPOutput
16 else if s_tok[5] == “if” then
17     Body = s_tok[6];
18     ASPBody = TransBody(Body);
19     fields = TransFields(s_tok[4]);
20     ASPHead = ASPHeadBase + “,” + fields + “).”;
21     ASPOutput = ASPHead + “:-” + ASPBody + “.”;
       Output: ASPOutput
```

3.3 EditSocACL

EditSocACL is a web-based tool for working with SocACL PBs. It is written using a combination of HTML, PHP, AngularJS, and SQL. The organisation of EditSocACL is summarised in Figure 3.2.

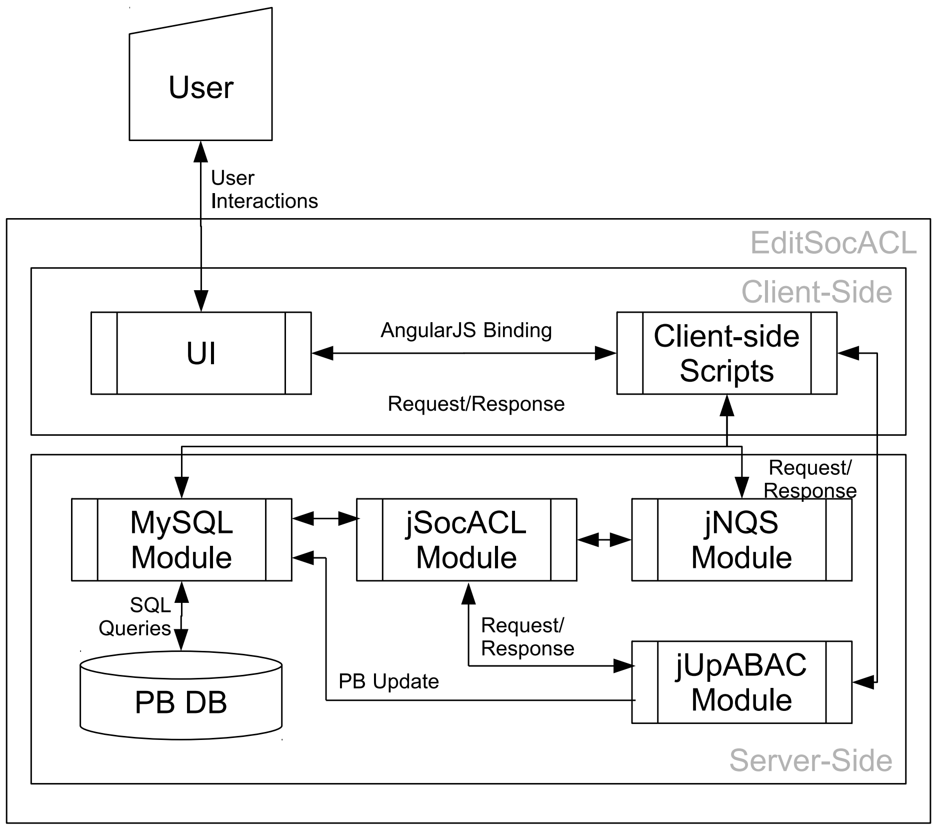


Figure 3.2: EditSocACL Flowchart.

Users interact with EditSocACL through a HTML-based UI. Dynamic portions of this interface are populated with information retrieved from the server-side components through client-side AngularJS scripts. These scripts make requests to the various server-side modules via POST. All of these models represent collections of PHP functions which interact with server-side assets.

As the name indicates, the `MySQL Module` is a collection of PHP functions responsible for interacting with EditSocACL's MySQL database. This database stores the user's SocACL PB and is accessed indirectly by the other server-side modules using the `MySQL Module`. Database interactions utilise the PHP PHP Data Objects (PDO) extension. Given PDO defines an interface independent of any specific database implementation conceivably EditSocACL could be easily modified for use with databases besides MySQL.

The `jSocACL Module` is used to translate SocACL PBs to ASP by having PHP invoke the Java tool jSocACL, presented in Section 3.1. The module requests a specific user's SocACL PB and translates it. This translation is then either displayed to the user, or is used by the `jNQS Module` or `jUpABAC Module`. It should be noted the translation is never stored. Every time there is a need for a translation of a user's SocACL PB the `jSocACL Module` is invoked. This is done rather than storing the translation because:

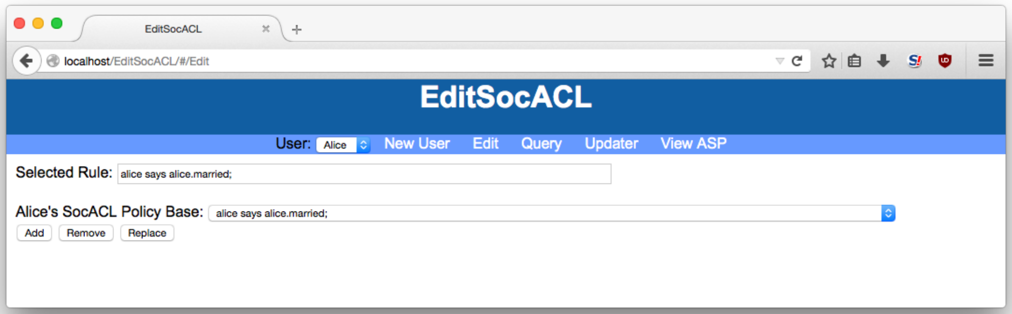
1. Performing the translation in this way is more indicative of the formalisms presented throughout this thesis
2. It simplified the design of EditSocACL.

The `jNQS Module` performs a query as per Chapter 4 using the Java prototype jNQS, presented later in Section 5.1, by requesting a translated PB from the `jSocACL Module`. To preform an update of the user's policy the `jUpABAC Module` also makes a request to the translation module. The rules of the PB's translation are mapped to the original SocACL rules such that once update candidate Δ is computed using the Java prototype jUpABAC, also presented later in Section 7.1, updates can then be applied to the `PB DB` using SQL queries.

3.3.1 Working with EditSocACL

This section outlines the usage of EditSocACL. It should be noted the screenshots and examples given in this section do not follow the continuity of the thesis' running case study.

3.3.2 UI Overview



Selected Rule: alice says alice.married;

Alice's SocACL Policy Base: alice says alice.married;

Figure 3.3: EditSocACL UI Layout.

Figure 3.3 shows the basic layout of EditSocACL's UI. This UI is centralised around the main menu found beneath the EditSocACL banner. The left-most item in this menu is a dropdown box for selecting a user currently in the EditSocACL system. To the right of this is a link which opens a form for adding new users to the system. The "Edit" form allows for direct editing of the selected user's SocACL PB for debugging purposes. "Query" directs the user to a form for performing queries. Similarly, "Updater" displays the interface for applying updates as per Chapter 6 to the selected user's PB. Finally the "View ASP" link simply displays the user's PB translated to ASP.

3.3.3 Debugging Tools

EditSocACL also provides a number of useful tools for debugging and experimenting with SocACL PBs.

Adding New Users

Following the "New User" link from the main menu takes the user to the form shown in Figure 3.4. Here a user can add new users to the system. To do this they ensure the "Manual Mode" option is selected. Then they provide the first and last name of the new user. Upon pressing the "Add" button EditSocACL creates a new user entry in its database, allowing for the new user to be selected using the "User" dropdown in the main menu.

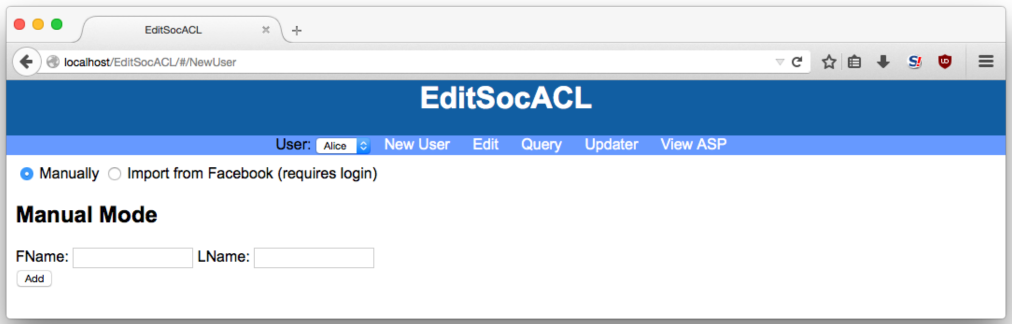
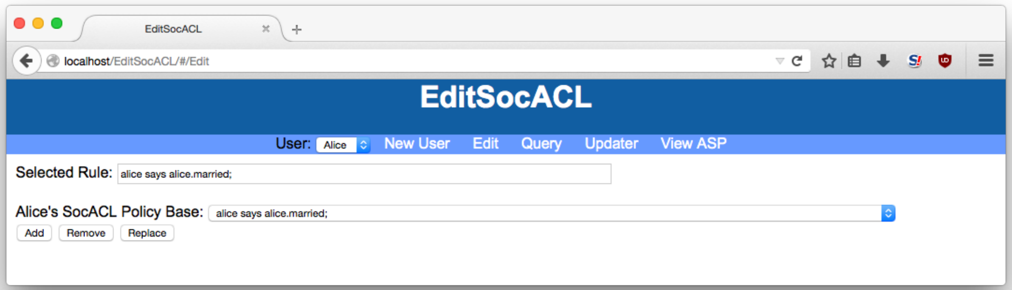


Figure 3.4: Adding a New User.

Direct Editing

For debugging and testing purposes EditSocACL allows for direct editing of the selected user's SocACL PB. Once a user is selected from the "User" dropdown menu that user's SocACL PB populates the Edit form. The "Selected Rule" box is an editable textbox containing the rule currently selected by the dropdown directly below it. As it can be seen in Figure 3.5, Alice's married attribute has been selected.

Buttons below the selected rule dropdown correspond to different edit operations. The "Add" button adds the contents of the "Selected Rule" textbox as a new rule to the selected user's PB. Note: this operation does *not* check if the new rule is syntactically correct. Conversely, the "Remove" button deletes the selected rules from the PB. The "Replace" operation replaces the currently selected rule with the contents of the textbox.

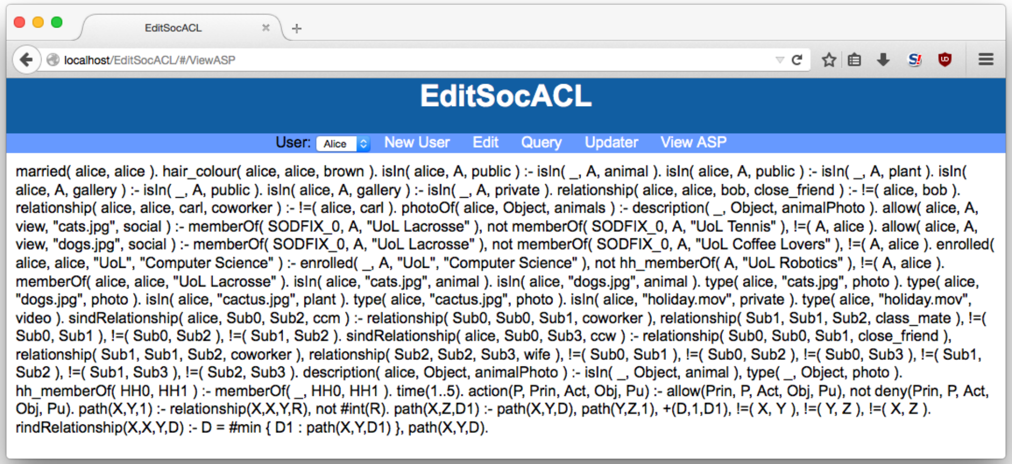


Selected Rule: alice says alice.married;

Alice's SocACL Policy Base: alice says alice.married;

Figure 3.5: Direct Editing.

Viewing a Translation



```
married( alice, alice ). hair_colour( alice, alice, brown ). isIn( alice, A, public ) :- isIn( _, A, animal ). isIn( alice, A, public ) :- isIn( _, A, plant ). isIn(
alice, A, gallery ) :- isIn( _, A, public ). isIn( alice, A, gallery ) :- isIn( _, A, private ). relationship( alice, alice, bob, close_friend ) :- !=( alice, bob ).
relationship( alice, alice, carl, coworker ) :- !=( alice, carl ). photoOf( alice, Object, animals ) :- description( _, Object, animalPhoto ). allow( alice, A,
view, "cats.jpg", social ) :- memberOf( SODFIX_0, A, "UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Tennis" ), !=( A, alice ). allow( alice, A,
view, "dogs.jpg", social ) :- memberOf( SODFIX_0, A, "UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Coffee Lovers" ), !=( A, alice ). enrolled(
alice, alice, "UoL", "Computer Science" ) :- enrolled( _, A, "UoL", "Computer Science" ), not hh_memberOf( A, "UoL Robotics" ), !=( A, alice ).
memberOf( alice, alice, "UoL Lacrosse" ). isIn( alice, "cats.jpg", animal ). isIn( alice, "dogs.jpg", animal ). type( alice, "cats.jpg", photo ). type( alice,
"dogs.jpg", photo ). isIn( alice, "cactus.jpg", plant ). type( alice, "cactus.jpg", photo ). isIn( alice, "holiday.mov", private ). type( alice, "holiday.mov",
video ). sindRelationship( alice, Sub0, Sub2, ccm ) :- relationship( Sub0, Sub0, Sub1, coworker ), relationship( Sub1, Sub1, Sub2, class_mate ), !=(
Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ). sindRelationship( alice, Sub0, Sub3, ccw ) :- relationship( Sub0, Sub0, Sub1, close_friend ),
relationship( Sub1, Sub1, Sub2, coworker ), relationship( Sub2, Sub2, Sub3, wife ), !=( Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub0, Sub3 ), !=( Sub1,
Sub2 ), !=( Sub1, Sub3 ), !=( Sub2, Sub3 ). description( alice, Object, animalPhoto ) :- isIn( _, Object, animal ), type( _, Object, photo ).
hh_memberOf( HH0, HH1 ) :- memberOf( _, HH0, HH1 ). time(1..5). action(P, Prin, Act, Obj, Pu) :- allow(Prin, P, Act, Obj, Pu), not deny(Prin, P, Act,
Obj, Pu). path(X,Y,1) :- relationship(X,X,Y,R), not #int(R). path(X,Z,D1) :- path(X,Y,D), path(Y,Z,1), +(D,1,D1), !=( X, Y ), !=( Y, Z ), !=( X, Z ).
rindRelationship(X,X,Y,D) :- D = #min { D1 : path(X,Y,D1) }, path(X,Y,D).
```

Figure 3.6: Viewing a SocACL Translation.

After selecting a user from the "User" dropdown menu following the "View ASP" link in the main menu provides a translation of the user's SocACL PB. For

Setting	**Description**
FRIEND_COUNT	The number of Friends to be considered in this PB.
FRIEND_COUNT_TYPE	The number of Friend types, e.g. Friend, Spouse, Coworker, etc. to be considered in this PB.
FRIEND_FLIM	Upper limit on many Friends each Friend has a relationship with.
FRIEND_TLIM	Upper limit many types of relationship each Friend has with another Friend, e.g. a Friend may consider someone else a Friend, Classmate, and Coworker.
FRIEND_INFER_COUNT	How many of the relationships considered are inferred.
FRIEND_INFER_MAX_BODY	The maximum number of SocACL terms which can be in any inferred relationship's Body.

Table 3.1: PolicyGen Relationship Configuration Settings

example, Figure 3.6 shows the translation of Alice's PB.

3.4 Experiments

The performance of jSocACL is evaluated by having it translate a collection of SocACL PBs of varying size. These experiments have been performed on a computer of the following specification: Intel Core i7 2.9GHz, 8GB RAM Apple MacBook Pro running OSX 10.10.3, and Java SE RE 1.6.0_37.

3.4.1 Automated Generation of SocACL PBs

Since the manual authoring of large PBs for the purposes of performance testing is tedious we have developed a utility called *PolicyGen* to automatically generate syntactically correct SocACL PBs. PolicyGen produces SocACL PBs based on the settings defined in a configuration file. These settings are outlined in Tables 3.1, 3.2, and 3.3. One notable limitation of PolicyGen is that it cannot generate aggregate terms, Section 2.7. This is due to technical difficulties encountered during PolicyGen's development and a lack of time committed to producing the utility.

Table 3.1 outlines the parameters which control how relationships are generated by PolicyGen. FRIEND_COUNT defines how many Friends are considered in the PB. For instance, if FRIEND_COUNT=1000 then PolicyGen generates

Setting	**Description**
ATTR_COUNT	The number of different attribute types.
ATTR_COUNT_VAL	The number of different values an attribute's field could possibly be.
ATTR_COUNT_VAL_MAX	Maximum number of field values a single attribute type can hold.
ATTR_INFER_COUNT	Number of inferred attributes.
ATTR_INFER_MAX_BODY	Maximum number of terms which a inferred attribute can have in its Body.
DESC_COUNT	Number of description definitions statements in the generated PB.
DESC_BODY_LIMIT	Maximum number of terms which can be used to form a description.

Table 3.2: PolicyGen Attribute Configuration Settings

1000 unique names for Friends. Similarly, FRIEND_COUNT_TYPE controls how many types of relationships are to be considered. Each Friend can be in FRIEND_TLIM types relationships with FRIEND_FLIM other Friends, where FRIEND_FLIM and FRIEND_TLIM are integer values. An important concept in PolicyGen's implementation is the Policy Authority (PA), a subject identifier denoting who "owns" the PB being generated. FRIEND_INFER_COUNT sets how many of the inferred relationships the PA holds with the Body size of up to FRIEND_INFER_MAX_BODY terms.

In order to generate rind and sind relationships PolicyGen creates a social graph using the setting presented in Table 3.1. For instance, rind relationships are generated by selecting a Friend some random distance "away" from the PA. These rind relationships are subsequently used to generate sind relationships.

PolicyGen prepares attributes and descriptions with respect to the settings in Table 3.2. ATTR_COUNT limits how many types or names of attributes are to be considered, while ATTR_COUNT_VAL determines how many different values an attribute field could take on. Furthermore, each attribute can have at most ATTR_COUNT_VAL_MAX number of fields. Similar to the relationship settings, ATTR_INFER_COUNT and ATTR_INFER_MAX_BODY limit the number of inferred attributes and how many terms form their Body. The number of descriptions generated for the PA is limited by DESC_COUNT, while DESC_BODY_LIMIT controls the maximum number of randomly selected terms which form the description.

The PA holds AUTH_COUNT of authorisation statements. Each statement's Body can contain between AUTH_BODY_MIN and AUTH_BODY_MAX number of randomly selected attributes, relationships, or descriptions generated by

Setting	Description
AUTH_COUNT	Total number of authorisation statements to be generated.
AUTH_BODY_MIN	Minimum number of terms in any authorisation's Body.
AUTH_BODY_MAX	Maximum number of terms in any authorisation's Body.
AUTH_OBJ_COUNT	Number of different objects considered when generating the PB.
AUTH_ACT_COUNT	Number of different actions considered when generating the PB.
PA_ATTR_COUNT	Number of attributes held by the Policy Authority.
PA_ATTRDISC_COUNT	Number of attributes disclosure rules held by the Policy Authority.
PA_ATTRDISC_BODY_MAX	Maximum number of terms the Body of a Attributes Disclosure Rules can contain.

Table 3.3: PolicyGen Authorisation and Policy Authority Configuration Settings

PolicyGen. These authorisations control access to AUTH_OBJ_COUNT different objects, and consider AUTH_ACT_COUNT types of actions.

Out of the large pool of attributes the PA is only assigned a subset containing PA_ATTR_COUNT attributes and another PA_ATTRDISC_COUNT sized set of attributes protected by an attribute disclosure rule. The Body can contain up to PA_ATTRDISC_BODY_MAX attributes, relationships, and descriptions.

3.4.2 Experiment: PolicyGen

To assess the performance of jSocACL it is tested against a set of SocACL PBs generated by PolicyGen. Each PB has been generated w.r.t. a different configuration file. The complete set of configuration settings can be found in Appendix D.1.

For each PB the number of ASP rules resulting from the translation is recorded. How much CPU time was taken to complete the translation of a single PB is also record in seconds. Table 3.4 summarise the results of these experiments. As it can be seen in Table 3.4 the number of rules in each generated PB varies dramatically, with the smallest PB containing 15 rules, and the largest containing 56963. Appendix D.1 shows configuration settings to generate PB Gen19 and Gen20. The subsequent PBs are absent from Table 3.4 as these two configurations caused PolicyGen to reach its timeout of 5 minutes.

In all of these experiments the ASP translation contains exactly four more

Config.	SocACL Statements	ASP Rules	Translation Time (s)
Gen01	15	19	0.11
Gen02	37	41	0.13
Gen03	89	93	0.18
Gen04	74	78	0.17
Gen05	137	141	0.20
Gen06	177	181	0.23
Gen07	401	405	0.25
Gen08	625	629	0.31
Gen09	977	981	0.36
Gen10	1263	1267	0.42
Gen11	1935	1939	0.70
Gen12	3748	3753	1.37
Gen13	8104	8108	3.48
Gen14	10868	10872	6.96
Gen15	17007	17011	16.88
Gen16	17237	17241	32.25
Gen17	34403	34407	129.82
Gen18	56963	56967	319.37

Table 3.4: jSocACL Performance Results

rules than the SocACL PB. This is expected as an increase of four indicates the inclusion of the UA rules.

Translation time remains similar for Gen01 to Gen12, with a notable increase at Gen13. Similarly, Gen13, Gen14, and Gen15 have similar translation times. Unusually, there is a large difference between the time taken to translate Gen15 and Gen16 despite there only being a difference of 230 statements. Upon closer analysis of the two configurations it appears this sharp increase in translation time is caused by Gen16 containing twice as many authorisation statements and inferred attributes as Gen15. This suggests the need for further improvement of certain sections of jSocACL's code, specifically the translation of authorisations and inferred attributes.

Overall these translation times are reasonable. Given the almost unnoticeable amount of time to translate PB with fewer than 17000 SocACL statements, it would be conceivable that these translations are performed on demand. On the other hand, PB with over 17000 take a noticeable amount of time to translate, with Gen17 and Gen18 taking 2 and 5 minutes respectively. For PBs of this size it would be more appropriate the perform the translations “offline” then store the ASP rules for later use.

3.5 Chapter Summary

This chapter has served as an introduction, discussion, and analysis of the Java implementation prototype of SocACL; jSocACL. Section 3.1.1 provided a general overview of the jSocACL system. The need for our prototype to perform an additional post-processing step was discussed in Section 3.1.2.

With jSocACL using code generated by ANTLR3, it is poorly human readable. For this reason, in Section 3.2 we have outlined the algorithms on which the ANTLR3 grammar was based, rather than the code itself.

jSocACL forms part of a larger system for the maintenance of SocACL PBs called EditSocACL. In Section 3.3 we have presented parts of EditSocACL closely tied to jSocACL.

The chapter concluded in Section 3.4 where we have evaluated the performance jSocACL using a series of experiments. These experiments have jSocACL translate a collection of different SocACL PBs. These PBs have been generated using our software tool PolicyGen, which we outlined in Section 3.4.1. The results of these experiments are discussed in Section 3.4.2.

Chapter 4

Negotiation Based Attribute-Based Access Control Policy Evaluation

4.1 Introduction

In access control scenarios resources are often held by principals known as *resource holders*. Resource holders specify who can and cannot access a resource by defining PBs such as those presented in Chapter 2. Requests or *queries* for access to these resources are presented by a *resource requester* to the holder. Holders evaluate these queries w.r.t. their own PBs and the attributes of the requester in order to assign a truth value to the query. This truth value denotes the access permission granted to the requester by the holder for a resource they own.

How the evaluation is performed varies between access control models. In the graph-based model proposed by Dhia [30] queries are treated as a node reachability check. Models such as MPAC and SocACL's basic queries (Section 2.10) utilise model checking. As previously discussed in Section 1.2.2 ABAC faces a number a challenges, some of which related to policy evaluation. In this chapter we tackle these challenges by outlining a novel approach to the evaluation of ABAC policies based on negotiations.

The chapter begins with an overview of the key concepts of this evaluation framework and preliminaries in Section 4.2. Section 4.3 uses these preliminaries to formalise PBs for use in this framework. Sections 4.4 and 4.5 continue by introducing messages exchanged by the requester and holder during policy evaluation. The chapter concludes in Section 4.6 where these messages are used to develop a formal protocol for this exchange.

4.2 Key Concepts and Preliminaries

During the evaluation of an ABAC policy a requester divulges his/her attributes to the holder to satisfy the holder's PB. This presents an interesting scenario where the requester is compromising his/her own privacy outcomes in exchange for potential access to a resource they desire. We describe it "potential access" because simply supplying the requested attributes does not imply being granted access.

The model for SocACL's basic queries, Section 2.10, assumes the resource holder has complete access to the requester's attributes during policy evaluation. However, the work by Li et al. [61] demonstrates that this assumption is neither robust nor optimal. Intuitively a requester places value on his/her attributes which they weigh against the value they place in the resource they wish to access.

Potentially the number of attributes revealed during policy evaluation could be minimised by having a requester gradually reveal more attributes as their query is rejected, but this introduces other issues. ABAC is particularly vulnerable to *attribute hiding attacks* [28]. Here an attacker withholds certain attributes to gain some advantage during policy evaluation. For example, Alice does not disclose certain parts of her driving history to get cheaper insurance. SocACL's support for NAF in its policies leaves the language especially vulnerable to these attacks as the NAF operator explicitly defines decision criteria which should be withheld.

The framework presented in this chapter tackles the above issues through the novel application of negotiations. ABAC policy evaluation holds parallels to the often illustrated buyer-seller negotiation case study. Here a buyer and seller exchange offers to reach a mutually acceptable outcome. These offers are based on opposing sets of preferences held by each agent. Intuitively, the buyer aims to get the best possible deal on a product which satisfies their wants and needs. Conversely, the seller aims to maximise profit by either maximising the product price or presenting alternative brands or models. For ABAC queries the requester wants access to some resource while revealing as few attributes as necessary. To establish trust the resource holder aims to gather as many of the requester's attributes mapped to access permissions.

In this chapter the agents engaged in a negotiation for access to some resource are referred to using the names *agent* and *opponent*. These names are relative to each other. For instance in a negotiation between Alice and Bob, Alice considers herself the agent while Bob is her opponent. Conversely, from Bob's perspective Alice is his opponent and he is the agent.

Unless stated otherwise, notation used throughout this chapter are consistent with previous definitions. For a set of atoms L and an atom $l \in L$, we denote the set of rules $\{ \ l_i. \mid l_i \in L \ \}$ by $\{L \leftarrow\}$. $Goal(l)$ denotes the constraint $\leftarrow$ *not* $l.$, while $Goal^-(l)$ denotes the constraint $\leftarrow l.$.

For a set of rules R, $Head(R)$ is the set of all atoms in the rule heads. $Body(R)^+$ denotes all positive atoms in rule bodies in R, and $Body(R)^-$ denotes all NAF atoms. Finally, $Body(R) = Body(R)^+ \cup Body(R)^-$ denoting the set of all body atoms in the rules in R.

4.3 Negotiation Knowledge Base

A negotiation is an exchange of offers between intelligent agents to reach a mutually acceptable conclusion on some goal. For this conclusion to be rational these offers need to be based on a consistent collection of facts, inference rules, and assumption held by each agent.

In the context of ABAC the goal denotes a query, while the agent's PB provides his/her facts and inference rules. Offers are also made w.r.t. to assumptions the agents makes about his/her opponent. This is because the agent and opponent should not have a complete understanding of the other's attributes. As such, the offers contain a "guess" at the other's actually attributes. For instance, Alice does not know Bob is enrolled in UoL Computer Science, but assumes he is when she presents Bob with an offer. Combining both the agent's PB and assumptions forms the triplet called a Negotiation Knowledge Base (NKB).

Definition 4.1 (***Negotiation Knowledge Base***)
$K = \langle \Pi, H^+, H^- \rangle$ is called an NKB where:

- Π is an ASP program representing a principals' policy which contains his/her attributes, authorisation, and attribute disclosure rules.
- H^+ is the set of atoms called *Positive Assumptions* which the agent safely assumes to be true such that $H^+ \cap Head(\Pi) = \emptyset$ and $H^+ \cap H^- = \emptyset$.
- H^- is the set of atoms called *Negative Assumptions* which the agent safely assumes to be false such that $H^- \cap Head(\Pi) = \emptyset$ and $H^- \cap H^+ = \emptyset$.

An NKB contains the PB of an agent expressed as the ASP program Π. Π consists of rules of the form (1.1). To improve the generality of the formalism it is presented in this chapter such that it does not consider SocACL until Section 5.1. This also eliminates the notation overhead of the translation in the query formalism.

Assumptions are attributes that an agent believes his/her opponent may or may not have. These are divided into *positive* and *negative assumptions.* H^+ is the set of *positive assumptions.* These are the attributes the agent believes his/her opponent holds and are derived from the atoms in $Body(\Pi)^+$, i.e. $A_1, \cdots, A_m$ in (1.1), found in Chapter 1. Conversely, H^- is the set of *negative assumptions*; attributes the agent believes his/her opponent does not hold. These are derived from the atoms in $Body(\Pi)^-$, i.e. $A_{m+1}, \cdots, A_n$ in (1.1). $H = H^+ \cup H^-$ is the set of both positive and negative assumptions.

Negotiations are said to conclude *successfully* if the requester is granted his/her request, while the negotiation concludes *unsuccessfully* if not. The conditions under which a negotiation concludes successfully or unsuccessfully will be highlighted as they arise throughout this chapter. First let us consider negotiations which involve *trivial policies.*

NKBs where $Body(\Pi) = \emptyset$ are said to contain a *trivial policy.* For NKBs which contain a trivial policy, when presented with a request for some resource, denoted by the *goal* atom g. If $\Pi \models g$ then the negotiation will always conclude successfully. Since assumptions are derived from $Body(\Pi)$ NKBs which contain a trivial policy also have an empty assumption set; $H^+ = \emptyset$, $H^- = \emptyset$. Conversely, NKBs with non-trivial policies, policies where $Body(\Pi) \neq \emptyset$, $(\Pi \cup \{H^+ \leftarrow\}) \models g$ must hold for a successful conclusion.

This chapter continues to develop the running case study introduced in Chapter 2. As previously mentioned for this chapter PBs of agents are ASP programs rather than SocACL PBs. However, the predicates of these programs adhere to the form of translated SocACL statements. Complete versions of these example programs can be found in Appendix B. Let us begin building the example for this chapter. Alice and Bob have the NKBs $K_{Alice} = \langle \Pi_{Alice}, H^+_{Alice}, H^-_{Alice} \rangle$ and $K_{Bob} = \langle \Pi_{Bob}, H^+_{Bob}, H^-_{Bob} \rangle$, respectively.

Example 4.1 (K_{Alice})

Π_{Alice} contains:

```
1 allow(alice, A, view, "cats.jpg", social) :- memberOf(_, A, "UoL
    Lacrosse"), not memberOf(_, A, "UoL Tennis"), A != alice.
2 allow(alice, A, view, "dogs.jpg", social) :- memberOf(_, A, "UoL
    Lacrosse"), not memberOf(_, A, "UoL Coffee Lovers"), A != alice.
3 enrolled(alice, alice, "UoL", "Computer Science") :- enrolled(_, A, "
    UoL", "Computer Science"), not memberOf(_, A, "UoL Robotics"), A !=
    alice.
4 memberOf(alice, alice, "UoL Lacrosse").
```

In the above lines 1 and 2 are authorisation rules. Alice allows others to view the image `cats.jpg` if they are a member of the University of Learning's Lacrosse club, but not a member of the UoL Tennis club. She also allows members of the Lacrosse club who are not also member of the Coffee club to view `dogs.jpg`. For both rules, Alice does not allow herself to be granted the permission. Line 3 is an attribute disclosure statement stating Alice will reveal her enrollment in the UoL Computer Science programme to others in the same programme who are not members of the Robotics club. On line 4 Alice asserts her Lacrosse club membership.

By grounding atoms in $\mathit{Body}(\Pi_{\mathit{Alice}})^+$ and $\mathit{Body}(\Pi_{\mathit{Alice}})^-$ w.r.t. her current opponent, Bob, Alice prepares her sets of assumptions. H^+_{Alice} contains:

```
memberOf(bob, bob, "UoL Lacrosse"), enrolled(bob, bob, "UoL", "Computer
  Science")
```

Alice assumes Bob is a member of the Lacrosse club and enrolled in UoL Computer Science. H^-_{Alice} contains:

```
memberOf(bob, bob, "UoL Coffee Lovers"), memberOf(bob, bob, "UoL Tennis
  "), memberOf(bob, bob, "UoL Robotics")
```

Alice hopes Bob is not a member of the Coffee, Tennis, or Robotics club.

Example 4.2 (K_{Bob})

Π_{Bob} contains:

```
memberOf(bob, bob, "UoL Lacrosse") :- enrolled(_, A, "UoL", _), A !=
  bob.
memberOf(bob, bob, "UoL Coffee Lovers") :- memberOf(_, A, "UoL Lacrosse
  "), A != bob.
enrolled(bob, bob, "UoL", "Computer Science").
```

Above shows the policy of Bob. He is a member of the UoL Lacrosse club and is willing to reveal this fact to anyone enrolled at UoL. Bob is also a member of the UoL Coffee club, revealing this to other Lacrosse club members. The final line is an attribute indicating that Bob is enrolled at UoL in Computer Science.

H^+_{Bob} contains:

```
1 enrolled(alice, "UoL", "Mathematics"), enrolled(alice, "UoL", "Computer
    Science"),
2 memberOf(alice, "UoL Lacrosse")
```

On line 1 of the above, Bob limits the courses considered by his "enrolled in any UoL course" rule to Mathematics and Computer Science. This is done for the practicality of the example. If we had included the full range of courses of a typical comprehensive university space would become a concern. As none of Bob's rules contain NAF atoms, he has no negative assumptions, thus $H^{-}_{Bob} = \emptyset$.

4.4 Proposals

Proposals are offers from an agent to their opponent over access to some resource, called the *goal*. Proposals take the form of a pair derived from the agent's NKB as follows.

Definition 4.2 (*Proposal*)
Let $K_A = \langle \Pi, H^+, H^- \rangle$ be A's NKB and atom g be the *goal*. H^+_* and H^-_* denote minimal subsets of H^+ and H^-, respectively, such that there exist the answer sets:

- $M^+ \in Ans(\Pi \cup \{H^+_* \leftarrow\} \cup Goal(g))$
- $M^- \in Ans(\Pi \cup \{H^-_* \cup (M^+ \cap H^+) \leftarrow\} \cup Goal^-(g))$

For which there is a set of atoms $S = (M^+ \cap H^+) \cup (M^- \cap H^-)$ called *support* which leads to a conclusion on g. The pair $\langle g, S \rangle$ is called a *proposal* for g by A w.r.t. K. The set of all proposals for g by A w.r.t. K is denoted by $\alpha(K, g)$.

S contains all assumptions which the agent needs to confirm with their opponent to reach a rational conclusion on the request for g. S is derived from the answer sets M^+ and M^-. M^+ contains a minimal subset of H^+ where the goal g holds w.r.t. K. M^- contains a minimal subset of H^- where g does *not* hold w.r.t. K and the positive assumptions in M^+. As such, S contains the negative assumptions which cause g to not hold when in the presence of the positive assumptions which would otherwise cause g to hold. There is no implied relationship between M^+ and M^-.

As discussed in Chapter 1, Crampton et al. [28] note that ABAC is particularly vulnerable to *attribute hiding attacks*. In these attacks a hostile opponent strategically withholds attributes to gain an advantage during the policy evaluation process. For example, a person with a poor driving history withholds certain details from their insurer to get a better price. Since PBs as defined in this chapter and in SocACL allow the use of the nonmonotonic operand "not" these attacks of are significant concern. Consider the following rule in Π_{Alice} from example 4.1:

```
allow(alice, A, view, "cats.jpg") :- memberOf(_, A, "UoL Lacrosse"),
  not memberOf(_, A, "UoL Tennis"), A != alice.
```

Here the rule explicitly states that not being a member of the Tennis club is advantageous for gaining access to `cats.jpg`. In effect the rule is advertising itself as a target for attribute hiding attacks. This scenario is considered by the definition of S. The definition of M^+ and M^- result in S presenting NAF literals, such as the Tennis club membership attribute, without its identifying "not" operand. This in turn obfuscates whether or not the opponent gains any advantage withholding a particular attribute. Moreover, rules which contain multiple NAF literals only one needs to be held by the opponent for the rule to not hold. With M^- defined such that it contains a minimal subset of H^- the number of negative assumptions revealed in S is at worst one per rule.

Example 4.3 ($\alpha(K_{Alice}, g)$)

For $\alpha(K_{Alice}, allow(alice, bob, view, \text{``}cats.jpg\text{''}, social))$, where K_{Alice} is from Example 4.1.

$$H^+_{Alice,*} = \{memberOf(bob, bob, \text{``}UoL\ Lacrosse\text{''})\}$$
$$H^-_{Alice,*} = \{memberOf(bob, bob, \text{``}UoL\ Tennis\text{''})\}$$

$M^+ =$

```
{ memberOf(alice, alice, "UoL Lacrosse"),
allow(alice, bob, view, "cats.jpg", social),
allow(alice, bob, view, "dogs.jpg", social),
memberOf(bob, bob, "UoL Lacrosse") }
```

While $M^- =$

```
{ memberOf(alice, alice, "UoL Lacrosse"),
allow(alice, bob, view, "dogs.jpg", social),
memberOf(bob, bob, "UoL Tennis"),
memberOf(bob, bob, "UoL Lacrosse") }
```

As such:

$$\begin{aligned} S =& \{memberOf(bob, bob, \text{“}UoL\ Lacrosse\text{”})\} \cup \\ & \{memberOf(bob, bob, \text{“}UoL\ Tennis\text{”})\} \\ =& \{memberOf(bob, bob, \text{“}UoL\ Lacrosse\text{”}), \\ & memberOf(bob, bob, \text{“}UoL\ Tennis\text{”})\} \end{aligned}$$

Resulting in $\alpha(K_{Alice}, allow(alice, bob, view, \text{“}cats.jpg\text{”}, social))$ containing the proposal:

```
< allow(alice, bob, view, "cats.jpg", social), { memberOf(bob, bob, "
  UoL Lacrosse"), memberOf(bob, bob, "UoL Tennis") } >
```

4.5 Responding to Proposals

When presented with a proposal $\langle g, S \rangle$ from their opponent an agent replies to it. As the proposal is effectively asking “do you hold attributes S for access to resource g?” this reply aims to answer these questions. This reply is formed by extending the proposal formalism.

S is extended so an agent can “echo” the attribute atoms in S they hold. A new set of atoms is also introduced called *rejections* which contains the attributes the agent declares the do not hold or cannot support. So far these extensions follow the work of Son and Sakama [77], but, as their work does not consider attribute disclosure statements a new utterance is introduced. These statements require a response such as “You want to know that? Well I need to ask this first.”. In other words, the agent needs to declare when an attribute holds conditionally. To accommodate these, a new pair called a *conditional assumption* is introduced as follows:

Definition 4.3 (***Conditional Assumption***)

Let $K_A = \langle \Pi_A, H_A^+, H_A^- \rangle$ and $K_B = \langle \Pi_B, H_B^+, H_B^- \rangle$ be the NKBs of agents A and B, respectively. Let $Q_B = \langle g, S \rangle$ be a proposal for g by B w.r.t. K_B. H_*^+ and H_*^- denote minimal subsets of H^+ and H^-, respectively, such that for every atom $sg_i \in S$ there are the answer sets:

- $M^+ \in Ans(\Pi \cup \{\ H_*^+ \leftarrow\ \} \cup Goal(sg_i))$

- $M^- \in Ans(\Pi \cup \{H^-_* \cup (M^+ \cap H^+) \leftarrow\} \cup Goal^-(sg_i))$

If there exists *no* M^+ or M^-, where $(M^+ \cap H^+) \cup (M^- \cap H^-) = \emptyset$, then sg_i is a *sub-goal* of Q_B w.r.t. K_A. Sub-goals form part of the tuple $\langle sg_i, SS_i \rangle$ called a *conditional assumption* where $SS_i = (M^+ \cap H^+) \cup (M^- \cap H^-)$ is a set of atoms, called *sub-support*, which leads to a conclusion on sg_i. The set of all conditional assumptions to Q_B w.r.t. K_A is denoted by $\gamma(K_A, Q_B)$.

Agent A attempts to generate a conditional assumption for every support atom in S by testing them against their NKB. Similar to proposals, Definition 4.2, this is achieved by extracting values from the answer sets M^+ and M^-. However, in this case M^+ and M^- contain atoms pertaining to the support of the atom sg_i. If for some sg_i there exists a M^+ or M^- where $(M^+ \cap H^+) \cup (M^- \cap H^-) = \emptyset$ then sg_i does not have a conditional assumption. In a sense, conditional assumptions are partial rule sharing with the pair containing the grounded rule head (the sub-goal) and a set of grounded body atoms (the sub-support).

Example 4.4 ($\gamma(K_{Bob}, Q_{Alice})$)

Consider the NKB K_{Bob} from Example 4.2 and let Q_{Alice} be the proposal from Example 4.3. $\gamma(K_{Bob}, Q_{Alice})$ contains the following conditional assumptions:

```
< memberOf(bob, bob, "UoL Lacrosse"), { enrolled(alice, alice, "UoL", "
  Mathematics") } >
< memberOf(bob, bob, "UoL Lacrosse"), { enrolled(alice, alice, "UoL", "
  Computer Science") } >
```

Line 1 comes about when:

$$H^+_{Bob,*} = \{enrolled(alice, alice, \text{“}UoL\text{”}, \text{“}Mathematics\text{”})\}$$
$$H^-_{Bob,*} = \emptyset$$

Resulting in $M^+ =$

```
{ enrolled(alice, alice, "UoL", "Mathematics"), allow(alice, bob, view,
   "cats.jpg", social), allow(alice, bob, view, "dogs.jpg", social),
  memberOf(bob, bob, "UoL Lacrosse"), enrolled(bob, bob, "UoL", "
  Computer Science") }
```

Such that, the SS for line 1:

$$SS_1 = \{enrolled(alice, alice, \text{“}UoL\text{”}, \text{“}Mathematics\text{”})\} \cup \emptyset$$

Similarly, for line 2 where :

$$H^+_{Bob,*} = \{enrolled(alice, alice, \text{“}UoL\text{”}, \text{“}Computer\ Science\text{”})\}$$
$$H^-_{Bob,*} = \emptyset$$

Results in $M^+ =$

```
{ enrolled(alice, alice, "UoL", "Computer Science"),
allow(alice, bob, view, "cats.jpg", social),
allow(alice, bob, view, "dogs.jpg", social),
memberOf(bob, bob, "UoL Lacrosse"),
enrolled(bob, bob, "UoL", "Computer Science") }
```

With these extensions to proposals in mind lets take the form $\langle g, S\rangle$ of proposals and expand it by adding a set of conditional assumptions and rejections. Doing so yields the 4-tuple $\langle g, \mathcal{S}, \mathcal{C}, \mathcal{R}\rangle$ called a *conditional proposal.*

Definition 4.4 (*Conditional Proposal*)

Let $K_A = \langle \Pi, H^+, H^-\rangle$ be agent A's NKB and $Q_B = \langle g, S\rangle$ be a proposal from agent B. H^+_* and H^-_* denote minimal subsets of H^+ and H^-, respectively, such that there exist the answer sets:

- $M^+ \in Ans(\Pi \cup \{\ H^+_* \leftarrow\ \} \cup Goal(g))$
- $M^- \in Ans(\Pi \cup \{H^-_* \cup (M^+ \cap H^+) \leftarrow\} \cup Goal^-(g))$

If $S \cap H^- \neq \emptyset$, then $\langle g, \mathcal{S}, \mathcal{C}, \mathcal{R}\rangle$ is a *conditional proposal* for Q_B w.r.t. K_A where:

- $SG = \{\ sg_i \mid \langle sg_i, SS_i\rangle\ \in \gamma(K_A, Q_B)\ \}$
- $\mathcal{S} = (M^+ \cap H^+) \cup (M^- \cap H^-) \cup (M^+ \cap S) \backslash SG$
- $\mathcal{C} = \{\ \langle sg_i, SS_i\rangle \mid SS_i \subseteq \mathcal{S}, \langle sg_i, SS_i\rangle\ \in \gamma(K_A, Q_A)\ \}$
- $\mathcal{R} = \{\ l \mid l \in S, l \notin M^+, l \notin SG\ \}$

SG is the set of all sub-goals for Q_B w.r.t. K_A. $\mathcal{S}$ is a set of atoms called *support* which excludes sub-goals and leads to a conclusion on g. $\mathcal{C}$ is the set of conditional assumptions used by A to reach a conclusion on g w.r.t. Q_B. $\mathcal{R}$ is a set of atoms called *rejections* denoting any support atoms from S which A cannot support.

If $S \cap H^- = \emptyset$ then the conditional proposal for Q_B w.r.t. K_A is $\langle \bot, \emptyset, \emptyset, \emptyset \rangle$ denoting an unsuccessful negotiation. The set of all conditional proposals for Q_B w.r.t. K_A is denoted by $\mathcal{A}(K_A, Q_B)$.

$\mathcal{S}$ is a direct extension of S from proposals. It retains its original purpose by containing the assumptions the agent needs to confirm with its opponent, while the addition of $(M^+ \cap S) \setminus SG$ results in the set also containing the attributes from S which the agent holds, excluding sub-goals. For each atom $sg_i \in S$ to have an associated conditional assumption in $\mathcal{C}$ there must be a conditional assumption $\langle sg_i, SS_i \rangle \in \gamma(K_A, Q_A)$ and $SS_i \subseteq \mathcal{S}$ indicating it was "active" in reaching a conclusion on g. $\mathcal{R}$ is a set of atoms from S which the agent does not hold.

Example 4.5 ($\mathcal{A}(K_{Bob}, Q_{Alice})$)

Consider the NKB K_{Bob} from Example 4.2 and let Q_{Alice} be the proposal from Example 4.3. $\mathcal{A}(K_{Bob}, Q_{Alice})$ contains the conditional proposals:

```
< allow(alice, bob, view, "cats.jpg", social),
{ enrolled(alice, alice, "UoL", "Computer Science") },
{ < memberOf(bob, bob, "UoL Lacrosse"), { enrolled(alice, alice, "UoL",
   "Computer Science") } > },
{ memberOf(bob, bob, "UoL Tennis") } >

< allow(alice, bob, view, "cats.jpg", social),
{ enrolled(alice, alice, "UoL", "Mathematics") } ,
{ < memberOf(bob, bob, "UoL Lacrosse"), { enrolled(alice, alice, "UoL",
   "Mathematics") } > },
{ memberOf(bob, bob, "UoL Tennis") } >
```

For all of the above conditional proposals:

$$\mathcal{R} = \{memberOf(bob, bob, \text{“}UoL\ Tennis\text{”})\}$$

This is because there exists no $H^+_{Bob,*}$ for $\mathcal{R}$ where:

$$memberOf(bob, bob, \text{“}UoL\ Tennis\text{”}) \in M^+$$

As such, in the above conditional proposals $\mathcal{C} \subset \gamma(K_{Bob}, Q_{Alice})$ (Example 4.4) such that every $\langle sg_i, SS_i \rangle \in \mathcal{C}$, $SS_i \subseteq \mathcal{S}$.

Just as with support atoms there will be conditional assumptions which an agent cannot support that must be rejected. This highlights a problem with Definition 4.4; it can generate conditional assumptions, but it does not take them into account.

To address both of these issues let us first consider how to apply conditional assumptions. So far assumptions from the sets H^+ and H^- are included in the ASP programs which produce M^+ and M^- by generating rules from the assumption atoms. Following similar logic, rules are constructed from the conditional assumptions so they can be considered in ASP programs.

Rules generated from the conditional and regular assumptions are likely to overlap causing the conditional assumption rules to become useless. To account for this the rules generated from conditional assumptions are prioritised when preparing the ASP programs. All assumptions are applied as a set of rules derived from the sets H and $\mathcal{C}$:

$$\begin{aligned} Assum(H,\mathcal{C}) = (\ \{H \leftarrow\} \backslash \{\ sg_i.\mid \langle sg_i, SS_i\rangle \in \mathcal{C}\ \}\ ) \cup \\ \{\ sg_i \leftarrow ss_1, \cdots, ss_k. \mid ss_j \in SS_i (j = 1, \cdots, k), \\ \langle sg_i, SS_i\rangle \in \mathcal{C}\ \} \end{aligned} \tag{4.1}$$

Equation (4.1) produces a set of rules derived from conditional assumptions which take priority over rules derived from an agent's assumption set H. As conditional assumptions are partially confirmed assumptions they are treated as being "more correct" than the agent's own assumptions. Now let us expand the definition of conditional assumptions to take themselves into account.

Definition 4.5 (***Conditional Proposal Ext.***)

Let $K_A = \langle \Pi, H^+, H^- \rangle$ and K_B be NKBs of agent's A and B respectively, while $Q_B = \langle g, \mathcal{S}, \mathcal{C}, \mathcal{R}\rangle$ be a conditional proposal for g by B w.r.t. K_B. H^+_* and H^-_* denote minimal subsets of H^+ and H^-, respectively, such that for every atom $sg_i \in S$ there are the answer sets:

- $M^+ \in Ans(\Pi \cup Assum(H^+_*, \mathcal{C}) \cup Goal(sg_i))$
- $M^- \in Ans(\Pi \cup Assum(H^-_* \cup (M^+ \cap H^+), \mathcal{C}) \cup Goal^-(sg_i))$

If there exists no M^+ or M^-, where $(M^+ \cap H^+) \cup (M^- \cap H^-) = \emptyset$ then sg_i is a sub-goal of Q_B w.r.t. K_A. Sub-goals form part of the tuple $\langle sg_i, SS_i\rangle$ called a *conditional assumption*. $SS_i = (M^+ \cap H^+) \cup (M^- \cap H^-)$, called sub-support which denotes a set of atoms which leads to a conclusion on sg_i. We denote the set of all conditional assumptions to Q_B w.r.t. K_A by $\Gamma(K_A, Q_B)$.

Furthermore, for any conditional assumption $\langle sg_i, SS_i \rangle \in \mathcal{C}$. If there exists no M^+ or M^-, where $(M^+ \cap H^+) \cup (M^- \cap H^-) = \emptyset$ then $\langle sg_i, SS_i \rangle$ is also a conditional assumption for Q_B w.r.t. K_A and is also in $\Gamma(K_A, Q_B)$.

Finally a *response* is defined as a special case of conditional proposals which has been derived w.r.t. an agent's NKB and a conditional proposal from his/her opponent.

Definition 4.6 (*Response*)
Let $K_A = \langle \Pi, H^+, H^- \rangle$ be the NKB of agent A and $Q_B = \langle g, \mathcal{S}, \mathcal{C}, \mathcal{R} \rangle$ be a conditional proposal from another agent B. As such there exists a NKB $K_A \oplus Q_B = \langle \Pi, H^+ \backslash \mathcal{R}, H^- \backslash \mathcal{R} \rangle$ denoting a NKB of agent A updated w.r.t. information provided by B. H^+_* and H^-_* denote minimal subsets of H^+ and H^-, respectively, such that there exist the answer sets:

- $M^+ \in Ans(\Pi \cup Assum(H^+_*, (\mathcal{C} \cap \Gamma(K_A \oplus Q_B, Q_B))) \cup Goal(g))$
- $M^- \in Ans(\Pi \cup Assum(H^-_* \cup (M^+ \cap H^+), (\mathcal{C} \cap \Gamma(K_A \oplus Q_B, Q_B))) \cup Goal^-(g))$

A's response to Q_B w.r.t. $K_A \oplus Q_B$ is the conditional proposal $\langle g, \mathcal{S}', \mathcal{C}', \mathcal{F} \rangle$ where:

- $SG = \{ sg_i \mid \langle sg_i, SS_i \rangle \in \Gamma(K_A \oplus Q_B, Q_B) \}$
- $\mathcal{S}' = (M^+ \cap H^+) \cup (M^- \cap H^-) \cup (M^+ \cap S) \backslash SG$
- $\mathcal{C}' = \{ \langle sg_i, SS_i \rangle \mid \langle sg_i, SS_i \rangle \in \Gamma(K_A \oplus Q_B, Q_B), SS_i \subseteq \mathcal{S} \} \cup \{ \langle sg_i, SS_i \rangle \mid \langle sg_i, SS_i \rangle \in (\mathcal{C} \cap \Gamma(K_A \oplus Q_B, Q_B)) \}$
- $\mathcal{F} = \{ l \mid l \in S, l \notin M^+, l \notin SG \}$

However, A's response to Q_B w.r.t. $K_A \oplus Q_B$ is the conditional proposal $\langle \bot, \emptyset, \emptyset, \emptyset \rangle$ if any of the following holds:

- $\mathcal{S} \cap H^- \neq \emptyset$
- $SG \cap H^- \neq \emptyset$
- $H \backslash \mathcal{R} = \emptyset$

We denote the set of all responses to Q_B w.r.t. $K_A \oplus Q_B$ by $\beta(K_A \oplus Q_B, Q_B)$.

Responses differ from conditional proposal by updating the agent's NKB w.r.t. their opponent's conditional proposal. $K_A \oplus Q_B$ results in a revised NKB which no longer considers rejected attributes to avoid re-asking already established facts.

M^+ and M^- are now computed using $\mathcal{C} \cap \Gamma(K_A \oplus Q_B, Q_B)$, a subset of $\mathcal{C}$ where agent A can agree with these conditional assumptions. For M^- we apply a minimal subset of H^- and the positive assumptions from M^+ using $H^-_* \cup (M^+ \cap H^+)$. SG is the set of all possible sub-goals for Q_B w.r.t. $K_A \oplus Q_B$ while $\mathcal{S}'$ remains unchanged from the definition of $\mathcal{S}$ for conditional proposals.

$\mathcal{C}'$ consists of new conditional assumptions and ones which can carry over from $\mathcal{C}$. $\{ \langle sg_i, SS_i \rangle \mid \langle sg_i, SS_i \rangle \in \Gamma(K_A \oplus Q_B, Q_B), SS_i \subseteq \mathcal{S} \}$ is the set of new conditional assumptions introduced to the negotiation. $\{ \langle sg_i, SS_i \rangle \mid \langle sg_i, SS_i \rangle \in (\mathcal{C} \cap \Gamma(K_A \oplus Q_B, Q_B)) \}$ are the conditional assumptions from $\mathcal{C}$ which the agent can accept. Finally, $\mathcal{F}$ is the set of all rejected attribute atoms from $\mathcal{S}$.

If $\mathcal{S}$ contains any elements from H^-, there is a $\langle sg_i, SS_i \rangle \in \mathcal{C}$ where $sg_i \in H^-$, or $H \backslash \mathcal{R} = \emptyset$, then the agent's response is the special conditional proposal $\langle \bot, \emptyset, \emptyset, \emptyset \rangle$ denoting an unsuccessful negotiation. This is because these three cases denote scenarios where a successful negotiation is no longer possible. If either $\mathcal{S} \cap H^- \neq \emptyset$ or $\{ \langle sg_i, SS_i \rangle \mid \langle sg_i, SS_i \rangle \in \mathcal{C}, sg_i \in H^- \} \neq \emptyset$ is true then the opponent has confirmed they hold a negative assumption. $H \backslash \mathcal{R} = \emptyset$ indicates a scenario where the agent has run out of assumptions, so no new response can be generated.

Theorem 1 *For any NKB $K = \langle \Pi, H^+, H^- \rangle$ which contains a non-trivial policy and any conditional proposal $Q = \langle g, \mathcal{S}, \mathcal{C}, \mathcal{R} \rangle$. If $\mathcal{R} = \emptyset$ then $K \oplus Q = K$.*

Proof 4.1 Given a NKB $K = \langle \Pi, H^+, H^- \rangle$ which contains a non-trivial policy and a conditional proposal $Q = \langle g, \mathcal{S}, \mathcal{C}, \mathcal{R} \rangle$ where $\mathcal{R} = \emptyset$, then as per Definition 4.6:

$$\begin{aligned} K \oplus Q &= \langle \Pi, H^+ \backslash \emptyset, H^- \backslash \emptyset \rangle & (4.2) \\ &= \langle \Pi, H^+, H^- \rangle & (4.3) \\ &= K & (4.4) \end{aligned}$$

Therefore, Theorem 1 holds.

$\square$

The above theorem illustrates an interesting situation. Since the conditional proposal results in no change to the agent's NKB it is possible that responses resulting from this NKB make no progress towards a conclusion on the goal.

Definition 4.7 (***Productive Responses***)

Let the NKB of agent A be $K_A = \langle \Pi, H^+, H^- \rangle$ and $Q_B = \langle g, \mathcal{S}, \mathcal{C}, \mathcal{R} \rangle$ be a conditional proposal from agent B to A. We say that A's response to Q_B, $Q_A \in \beta(K_A \oplus Q_B, Q_B)$ is *productive* if at least one of the follow conditions holds:

1. $K_A \oplus Q_B \neq K_A$
2. $\mathcal{C} \subseteq \Gamma(K_A \oplus Q_B, Q_B)$
3. $Q_A = Q_B$
4. $\langle \bot, \emptyset, \emptyset, \emptyset \rangle \in \beta(K_A \oplus Q_B, Q_B)$

Productive responses are responses which make progress towards a conclusion on the goal. This progress is characterised by any one of the above conditions. Condition 1 has it so Q_A is the product of an updated NKB. 2 has the response accept all of the currently active conditional assumptions. Conditions 3 and 4 relate to negotiation success and failure, respectively, as we show later in Section 4.6.

Example 4.6 ($\beta(K_{Alice} \oplus Q_{Bob}, Q_{Bob})$)

Again considering the running Example. Let Q_{Bob} be line 1 from Example 4.5, such that $\beta(K_{Alice} \oplus Q_{Bob}, Q_{Bob})$. $K_{Alice} \oplus Q_{Bob} =$

$$\langle \Pi_{Alice}, H^+_{Alice} \backslash \{memberOf(bob, bob, "UoL\ Tennis")\},$$
$$H^-_{Alice} \backslash \{memberOf(bob, bob, "UoL\ Tennis")\} \rangle$$

Since Alice has been able to accept all of the conditional assumptions Bob has introduced into the negotiation she includes them in her response. She also introduces a new conditional assumption relating to her enrolment in the UoL Computer Science programme. Therefore, $\beta(K_{Alice} \oplus Q_{Bob}, Q_{Bob})$ contains:

```
<{allow(alice, bob, view, "cats.jpg", social)},
{memberOf(bob, bob, "UoL Robotics"), enrolled(bob, bob, "UoL", "
  Computer Science")},
{<{memberOf(bob, bob, "UoL Lacrosse")}, {enrolled(alice, alice, "UoL",
  "Computer Science")}>, <{enrolled(alice, alice, "UoL", "Computer
  Science")},{memberOf(bob, bob, "UoL Robotics"), enrolled(bob, bob, "
  UoL", "Computer Science")}>},
{}>
```

4.6 Negotiation

The negotiation follows a "I go, you go" model where the two agents take turns to exchange offers formalised as conditional proposals and responses until they reach some conclusion on the request.

Definition 4.8 (***Negotiation***)

Let agents A and B have the NKBs K_A and K_B, respectively. Negotiation over a request for g by B to A, starting with B, is a possibly infinite sequence of conditional proposals $w_1, \cdots, w_n$ where:

- $w_1 = \langle g, \emptyset, \emptyset, \emptyset \rangle$
- $w_i = \langle g, \mathcal{S}_i, \mathcal{C}_i, \mathcal{F}_i \rangle$
- $w_{i+1} \in \beta(K_{i+1}, w_i)$ where for every $i > 1$:
  - $K_0 = K_A$ and $K_{2k+2} = K_{2k} \oplus w_{2k+1}$ for $k \geq 0$,
  - $K_1 = K_B$ and $K_{2k+1} = K_{2k-1} \oplus w_2$ for $k > 0$.

A negotiation is said to have concluded *unsuccessfully* when $w_i = \langle \perp, \emptyset, \emptyset, \emptyset \rangle$. The negotiation has concluded *successfully* when $w_i \in \beta(K_i, w_i)$ and $w_i \in \beta(K_{i+1}, w_i)$. However, the negotiation is *infinite* if it concludes neither successfully nor unsuccessfully.

Negotiations can conclude successfully, unsuccessfully, or never conclude. Successful conclusions occur when both agents can accept the same conditional proposal, granting the request for g. The negotiation is unsuccessful if $w_i = \langle \perp, \emptyset, \emptyset, \emptyset \rangle$, the special response denoting the agent cannot continue this negotiation. If a negotiation concludes neither successfully nor unsuccessfully, then the negotiation is infinite. An infinite negotiation would occur when none of the responses w_i are productive as per Definition 4.7.

Definition 4.9 (***Productive Negotiations***)

A negotiation consisting of the sequence $w_1, \cdots, w_n$ of responses is said to be a *productive negotiation* if every $w_i (1 \leq i \leq n)$ is a productive response.

Proof 4.2 As per Definition 4.8 a negotiation can either conclude successfully or unsuccessfully. Consider the conditions under which a response can be productive by Definition 4.7. A successful negotiation where each w_i is productive under condition 1, such that the NKB:

$$\begin{aligned} K_{2k+2} &= K_{2k} \oplus w_{2k+1} \\ &= \langle \Pi_{2k}, H^{+}_{2k}, H^{-}_{2k} \rangle \oplus \langle g, \mathcal{S}_{2k+1}, \mathcal{C}_{2k+1}, \mathcal{R}_{2k+1} \rangle \\ &= \langle \Pi_{2k}, H^{+}_{2k} \backslash \mathcal{R}_{2k+1}, H^{-}_{2k} \backslash \mathcal{R}_{2k+1} \rangle \end{aligned} \tag{4.5}$$

Would eventually result in $H = \emptyset$, resulting in $\langle \perp, \emptyset, \emptyset, \emptyset \rangle \in \beta(K_{2k+2}, w_{2k+1})$, denoting an unsuccessful negotiation as per Definition 4.6. Therefore, a negotiation consisting of w_i's productive under condition 1 will eventually result in an unsuccessful negotiation.

Conversely, a productive negotiation consisting only of w_i's productive under condition 2 must conclude successfully since all conditional assumptions are acceptable under both agents, hence no rejections.

If all the w_i's in a negotiation are productive under conditions 3 and 4, as per Definition 4.7, then the negotiation must be finite as the conditions correspond to a successful or unsuccessful negotiation.

$\square$

With the above proof in mind we consider the following theorem.

Theorem 2 *If a negotiation is productive, then it is finite.*

Proof 4.3 For a negotiation to be finite it must conclude. Clearly, under Definition 4.9 and Proof 4.2 a productive negotiation must conclude. As such, the negotiation must also be finite. Therefore, Theorem 2 holds.

$\square$

4.7 Chapter Summary

In this chapter we presented a formalism for evaluating ABAC policies based on negotiation. Section 4.2 provided an overview of this formalism's key concepts along with preliminary notation. Using this notation in Section 4.3 we formalised the representation of PBs in our query framework. Sections 4.4 and 4.5 served to build a formal understanding of the messages exchanged by the negotiating principals. These messages are utilised in Section 4.6 to define a protocol for how these messages are generated and exchanged, and how a decision on the query is ultimately decided.

Chapter 5

Implementation of Negotiation Based Queries

5.1 Introduction

In this chapter we introduce and discuss jNQS (Negotiation Query System in Java), a prototype implementation of the negotiation based query outlined in Chapter 4.

5.1.1 Technical Overview

jNQS utilises DLVWrapper v4.2, a collection of Java interfaces for the ASP solver DLV. Figure 5.1 summarises the interaction of jNQS's various components.

Figure 5.1 shows self contained modules representing the agent and opponent. The modules exchange conditional proposals and responses via a negotiation co-ordinator. Both the agent and opponent take the PB of their respect principal as input. The opponent takes the query as an additional input. This data is stored within each agent/opponent module to form their respective NKBs.

The NKBs are used by the responder module to construct the ASP programs passed to the ASP solver to generate answer sets. These answer sets are then used by the responder to build conditional proposals or responses.

5.1.2 Java Classes of jNQS

The components illustrated in Figure 5.1 are represented in jNQS by the Java classes in this section. As each class has fairly standard "setter", "getter", display, and overridden `toString` methods they are omitted for brevity.

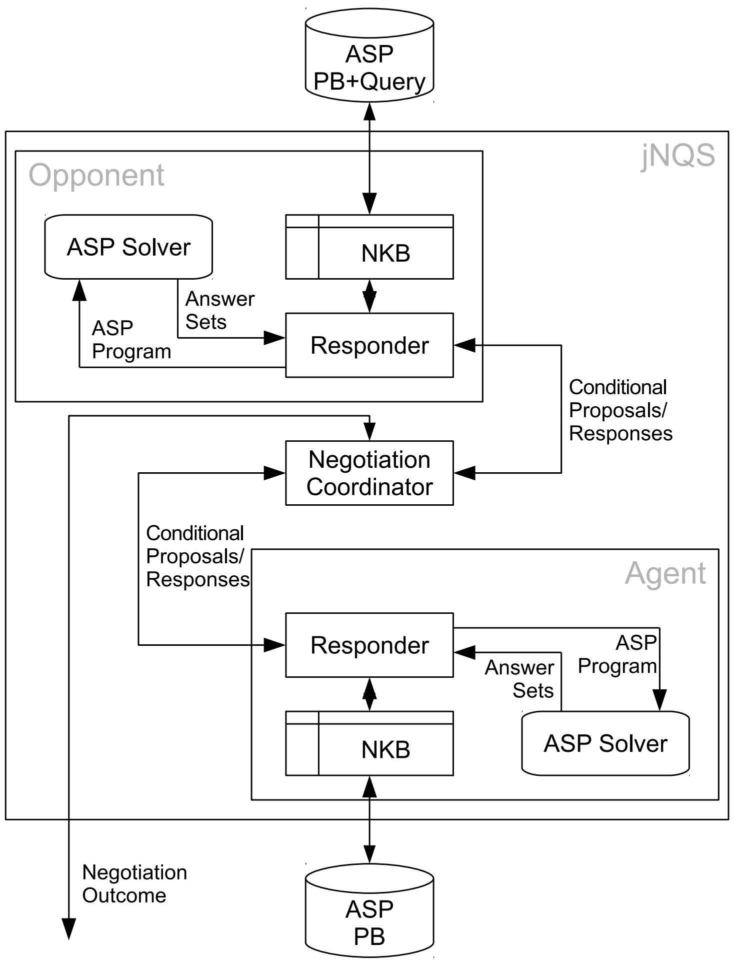


Figure 5.1: Negotiation Prototype Flowchart.

Class: `NKB`

The `NKB` class implements the NKB formalism, Definition 4.1. This is done through the class's properties:

- `Set<String> Pi`
- `Set<String> HPlus`
- `Set<String> HMinus`

The above `Set<String>`s correspond to Π, H^+ and H^- from Definition 4.1 respectively. To support these three properties the class also provides the following method:

- `void oplus(ConditionalProposal)`

`oplus(ConditionalProposal)` implements $K \oplus Q$ of the Response definition, Definition 4.6. This method takes an offer from the opposing agent, encoded as an instance of the `ConditionalProposal` class which is introduced later in this section. Though the logic implemented by this method does not form part of Definition 4.1, it is included here to simplify jNQS's implementation.

Class: CondAssum

Conditional assumptions, Definition 4.3, are implemented using the `CondAssum` class. This class has two notable properties:

- `Set<String> SG`
- `Set<String> SS`

`SG` implements the sub-goal sg_i from Definition 4.3, while `SS` corresponds to SS from the same Definition. Similar to `NKB`, the `CondAssum` class differs to the formalism on which it is based. Definition 4.3 includes the logic for the function $\Gamma()$ which denotes the set of all conditional assumptions. However, $\Gamma()$ is not implemented in 4.3, but rather in the `Agent` class.

Class: ConditionalProsposal

The `ConditionalProposal` class implements its namesake from Definition 4.4 along with responses of Definition 4.6. This class acts as a data store by having the following properties:

- `Set<String> G`
- `Set<String> S`
- `Set<CondAssum> CS`
- `Set<String> R`

Each of the above corresponds to their namesake from the Definitions 4.4 and 4.6 with a few exceptions. `G`, `S`, and `CS` correspond, respectively, to $g, \mathcal{S}$ and $\mathcal{CS}$. `R` is the exception corresponding to *both* $\mathcal{R}$ and $\mathcal{F}$ depending on the usage.

`ConditionalProposal`'s implementation differs from the formalism by having `G` be a set, rather than a single atom. Though this suggests that jNQS can accommodate a set a goals this is not the case. `G`'s definition is an artifact from the existing code on which all our pair and n-tuple code is based.

Class: Agent

Agents and opponents are implemented by, as illustrated in Figure 5.1, distinct instances of the same class: `Agent`. `Agent` has the following properties:

- `NKB myNKB`
- `TYPE Role`
- `Set<Set<String>> HpStar`
- `Set<Set<String>> HmStar`
- `Set<Set<String>> HStar`

`myNKB` is the NKB of an instance of the `Agent` class. `TYPE Role` is an enumeration which denotes which `Agent` agent instigated the negotiation, in turn determining the order of the offer exchange.

Definitions 4.3, 4.4, and 4.6 call for the use of minimal subsets of the H^+ and H^-. This is implemented by computing the powersets of H^+ and H^-, ordering them from smallest cardinality to largest, then applying them to our answer set computation in the same order. As this process may become time consuming, the results of computing the powersets are stored in `HpStar` and `HmStar`. `HStar` is simply a union of these results. The implementation also does not attempt to compute all answer sets for a given conditional proposal. Instead jNQS generates responses based on the first viable answer set.

Class `Agent` implements the vast majority of the negotiation logic from this chapter using the following methods:

- `ConditionalProposal reponse(ConditionalProposal)`
- `Set<CondAssum> Gamma(ConditionalProposal)`
- `Set<ConditionalProposal> Beta(ConditionalProposal)`
- `Set<String> MPlus(Set<String>, Set<String>)`
- `Set<String> MMinus(Set<String>, Set<String>, Set<String>)`
- `Set<Set<String>> Ans(Set<String>)`

Method `response()` prepares and returns a `ConditionalProposal` denoting a response. It does so by calling the private methods `Gamma()` and `Beta()`. These methods implement their respective namesakes from Definitions 4.3, 4.4, and 4.6 (`Beta()` implements both $\mathcal{A}()$ and $\beta()$).

The answer sets M^+ and M^- are generated by methods `MPlus()` and `MMinus()`. Both methods interact with the ASP solver DLV through the DLVWrapper API. These DLV interactions are performed via the `Ans()` method. `Ans()` takes a set of strings representing a set of rules which form an ASP program and prepares it for use with DLVWrapper. DLVWrapper uses a special class to return answer sets called `Model`. Before returning this answer set `Ans()` converts them from the `Model` class to a `Set<Set<String>>`.

Class: Negotiation

The `Negotiation` class implements Definition 4.8, illustrated in Algorithm 11. This class acts as the coordinator between the agent and opponent. `Negotiation` has the following properties:

- `Agent agent`
- `Agent opponent`
- `Set<String> G`
- `int rlimit`

`agent` and `opponent` are instances of the `Agent` class representing the negotiation agent and opponent. `Set<String> G` keeps a negotiation wide record of the goal. `rlimit` defines the number of rounds the negotiation is limited to ensure termination. A negotiation defaults to unsuccessful on error or when `rlimit` is reached.

The `Negotiation` class has a single method; `begin()`. This method initiates and coordinates the negotiation w.r.t. the constraints from Definition 4.8 through calling each `Agent`s public method: `reponse()`. As it can be seen in Algorithm 11, `begin()` also tests whether the negotiation was successful or unsuccessful, based on the conditions outlined previously in this chapter.

5.1.3 Queries using EditSocACL

As with jSocACL in Chapter 2, jNQS can be interacted with via the EditSocACL GUI. By following the “Query” link in EditSocACL takes the user to the form shown in Figure 5.2. Using the two dropdown menus “Agent” and “Opponent” the user selects which PB will function as the agent and the opponent. Below these dropdowns the user also must manually provide the assumptions used in the negotiation. Since manually providing the assumptions is far from ideal the automated generation of assumptions is noted as a consideration for future work. Beneath the Opponent’s assumptions text area, they provide their query that will be presented to the Agent. To start the query press button labelled “Do it!”. Once the negotiation is complete the results is displayed below the “Output” header.

Algorithm 11: Negotiation Algorithm

```
1  Let $K_{Ag}$ be the NKB of the resource holder.
2  Let $K_{Op}$ be the NKB of the resource requester.
3  Let $Q$ be the query as a conditional proposal.
4  Let $RoundLim$ be the negotiations max number of rounds.
5  $Q_{Op} = Q$
6  $OpAccept = false$
7  $round = 2$
8  while $round \neq RoundLim$ do
9      if $round$ mod $2 = 0$ then
10         $K_{Ag} = $ OPlus( $K_{Ag}$, $Q_{Op}$ )
11         $Q_{Ag} = $ Response( $K_{Ag}$, $Q_{Op}$ )
12         if $Q_{Ag} = Q_{Op}$ and $OpAccept = true$ then
13             Successful Negotiation
14         else
15             $OpAccept = false$
16     if $round$ mod $2 \neq 0$ then
17         $K_{Op} = $ OPlus( $K_{Op}$, $Q_{Ag}$ )
18         $Q_{Op} = $ Response( $K_{Op}$, $Q_{Ag}$ )
19         if $Q_{Op} = Q_{Ag}$ then
20             $OpAccept = true$
21         else
22             $OpAccept = false$
23     if $Q_{Ag} = \langle \perp, \emptyset, \emptyset, \emptyset \rangle$ or $Q_{Op} = \langle \perp, \emptyset, \emptyset, \emptyset \rangle$ then
24         Unsuccessful Negotiation
25     $round = round + 1$
26 Unsuccessful Negotiation
```

5.2 Experiments

The performance of jNQS has been evaluated using experiments based on the running case study of Alice and Bob, the example used in Son and Sakama [77], and the collection of PBs generated using PolicyGen from Section 3.4.1.

Experiments using the Alice and Bob case study, and the Son and Sakama example have been run on a computer of the following specifications; Intel Core i7 2.9GHz, 8GB RAM MacBook Pro running OSX 10.10.3, Java RE 1.6.0_37 and DLV release (2012-7-12). The experiments measure the average CPU time taken for a negotiation to reach a conclusion on some query w.r.t. the NKBs of the agent and opponent.

Time taken by each negotiation is measured using the Java `Main.getCpuTime()` call before and after a negotiation then taking the difference. For the Alice and Bob, and Son and Sakama experiments an average time is calculated by repeat-

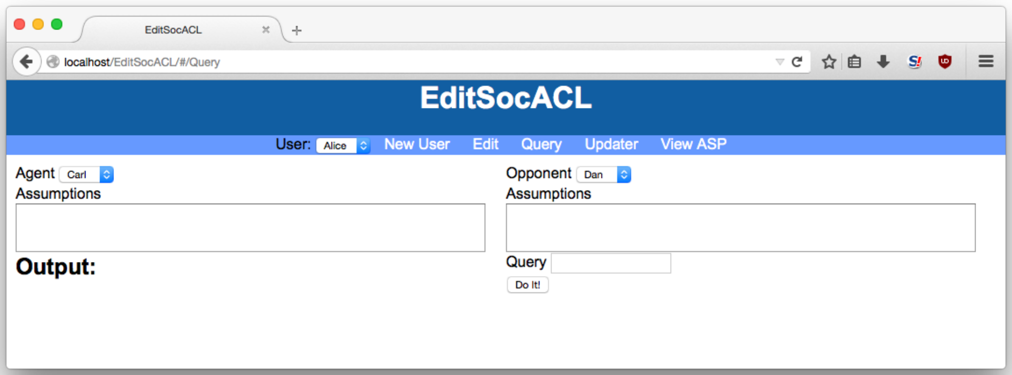


Figure 5.2: Query.

Experiment	Concludes	Time (ms)	Rounds
A+B 1	Successfully	263	8
A+B 2	Successfully	48	5
A+B 3	Unsuccessfully	89	2
A+B 4	Unsuccessfully	6	2
S+S 1	Successfully	681	10
S+S 2	Successfully	63	4
S+S 3	Unsuccessfully	10	2
S+S 4	Unsuccessfully	41	4

Table 5.1: Experiment Results.

ing the same query and NKB combination 1000 times. This averaging has not been done for the PolicyGen experiment given the very large size of some of the PBs. The time reported by `Main.getCpuTime()` is converted from nanoseconds to milliseconds for readability.

To gauge the complexity of an experiment set the ASP grounder *gringo* [41] is used to report the number of grounded rules in the program $\Pi \cup \{H^+ \leftarrow\}$ where Π and H^+ are the policy and positive assumptions of some principals' NKB.

Experiments involving the PolicyGen PBs have been performed differently to account for the large size of some of these PBs. Firstly, the experiments have been run on a different computer which has the following specifications: Intel Core i7 4.4GHz, 16GB RAM running Linux Mint 17.2 Rafaela, Java RE 1.6.0_37, and DLV release (2012-7-12). Secondly, these experiments do not use the PolicyGen PBs directly, instead they are used to extend the rules used in the previous experiments. The reasoning for both of these decisions and other details are explained later in Section 5.2.3.

5.2.1 Experiment: Alice and Bob

These experiments utilise the NKBs of Alice and Bob which we introduced in Examples 4.1 and 4.2. Four different queries are applied to these NKBs. A summary of the experiments and results is shown in Figure 5.1. Each query was applied already knowing if the negotiation would conclude successfully or unsuccessfully. Both experiment A+B 1 and A+B 2 conclude successfully, while A+B 3 and A+B 4 are unsuccessful.

A+B 1 has Bob query Alice `allow(alice,bob,view,"cats.jpg",social)` and involves one conditional assumption.

A+B 2 has Alice query Bob `enrolled(bob,bob,"UoL","Computer Science")` and involves no conditional assumptions.

A+B 3 has Bob query Alice `allow(alice,bob,view,"dogs.jpg",social)` and concludes unsuccessfully because Bob holds one of Alice's negative assumptions.

A+B 4 has Alice query Bob `allow(bob,alice,view,"fish.jpg",social)` and concludes unsuccessfully because Bob holds no such rule.

Grounding $\Pi_{Alice} \cup \{H^{+}_{Alice} \leftarrow\}$ finds the program contains 6 grounded rules, while $\Pi_{Bob} \cup \{H^{+}_{Bob} \leftarrow\}$ contains 8, giving a total of 14 grounded rules involved in these negotiations. As seen in Table 5.1 the queries which involved conditional assumptions (A+B 1, A+B 3) took longer in terms of time than ones that did not (A+B 2, A+B 4). This is to be expected as per Definition 4.6 for each response a set of conditional assumptions is also computed. However, in all cases the query was resolved within an acceptable time frame.

5.2.2 Experiment: Son and Sakama's Example

These experiments have been adapted from the buyer/seller example presented by Son and Sakama [77]. Since the framework presented in this chapter is based on the work Son and Sakama [77] it is appropriate to test whether or not the system can still support its originally intended use case. Given the differences between the policy language used in this chapter and Son and Sakama's [77] example the experiment takes liberties with the source material.

Example 5.1 ***(Seller NKB*** $\langle \Pi_{Seller}, H^{+}_{Seller}, H^{-}_{Seller} \rangle$)

$\Pi_{Seller} =$

```
whole_sale_customer :- registered.
student_customer :- student.
senior_customer :- age(A), A >= 65.
high_pr.
```

```
 5 low_pr :- senior_customer.
 6 low_pr :- student_customer, good_credit.
 7 low_pr :- student_customer, pay_cash.
 8 lowest_pr :- whole_sale_customer, quantity(A), A >= 100.
 9 make( A ) :- product( A, _, _).
10 madeIn( A ) :- product( _, A, _ ).
11 colour( A ) :- product( _, _, A ).
12 product("Top Lacrosse", "France", yellow) ∨ product("Lacrosse Tech", "
     Austria", blue) ∨ product("Ball-o-Rama", "New Zealand", yellow) :-
     high_pr, not low_pr.
13 product("Lacrosse Tech", "Austria", blue) :- low_pr, not lowest_pr.
14 product("Ball-o-Rama", "New Zealand", yellow) :- lowest_pr.
```

Π_{Seller} is the policy of the Seller. On Lines 1 to 3 the Seller groups certain attributes to form customer types. Lines 4 to 8 assign different pricing levels to these various types of customer. Lines 9 to 11 are rules that extract certain attributes from larger product descriptions.

The remaining lines represent the Seller's product database and use disjunctive rule heads to emulate choice. Though the use of disjunctive heads is at odds with our policy language, Definition 1.1, we make the concession in this case as a technique to encode preference. Since *Disjunctive Logic Programs* (DLPs) are at least as complex as Normal Logic Program (NLP)s [60] this change in policy language only results in less favourable conditions for our prototype. $H^{+}_{Seller} =$

```
1 { registered, student, age(65), good_credit, pay_cash, quantity(100) }
```

In this example the Seller only has positive assumptions despite their policy containing NAF atoms. This is done as another technique to capture the pricing preferences of the agent. As jNQS does not support automated generation of assumptions they are provided manually, highlighting more problems with this approach. It fails at capturing rules bodies such as `age(A), A = 65` and `quantity` as it would either require an infinite number of entires or, ideally, be represented using an aggregate operator.

Example 5.2 (***Buyer NKB*** $\langle \Pi_{Buyer}, H^{+}_{Buyer}, H^{-}_{Buyer} \rangle$)

$\Pi_{Buyer} =$

```
1 age(25). student. pay_cash. quantity(1).
2 sale :- make( "Top Lacrosse" ), madeIn( "France" ), not color( blue ),
    high_pr.
3 sale :- make( "Lacrosse Tech" ), not madeIn( "Australia" ), low_pr.
```

```
4 sale :- make( "Ball-o-Rama" ), color( tangerine ), low_pr.
5 sale :- make( "Econocrosse" ), lowest_pr.
```

Above shows the policy the of the Buyer. Line 1 contains their attributes. The Buyer encodes their purchase preferences with the rules on lines 2 to 5, where they are willing to accept a `sale` if a certain combination of product attributes and price are met.

$H^+_{Buyer} =$

```
{ make("Top Lacrosse"), make("Lacrosse Tech"), make("Ball-o-Rama"),
  make("Econocrosse"),
madeIn("France"), high_pr, low_pr, lowest_pr, color(tangerine) }
```

$H^-_{Buyer} =$

```
{ madeIn("Australia"), color(blue) }
```

Table 5.1 contains the experiment results. Experiment S+S 1 has the Seller query the Buyer `sale` to establish which combinations a product attributes the Buyer will accept at certain price points and the conditions under which the Seller will offer these prices. Experiment S+S 2 has the Seller query the Buyer `student` with the Seller attempting to establish if the Buyer is a student. Experiment S+S 3 has the Seller attempting to establish if the Buyer has good a credit rating with the query `good_credit`, which they do not. Experiment S+S 4 has the Buyer querying the Seller `whole_sale_customer` as the Buyer attempts to find out if the Seller considers them a whole sale customer, which they are not.

Grounding $\Pi_{BuyerPB} \cup \{H^+_{BuyerPB} \leftarrow\}$ finds the program contains 14 grounded rules, while $\Pi_{SellerPB} \cup \{H^+_{SellerPB} \leftarrow\}$ contains 20, giving a total of 34 grounded rules. It can be seen in Table 5.1 that these results are consistent with the Alice and Bob experiments. Queries which involve conditional assumptions take longer than those that do not. The time taken to resolve the query `sale` is significantly higher than any other query in these experiments, despite the number of rounds being comparable to the slowest query in the Alice and Bob experiment. Given that the agent and opponent in this example have conflicting rules relating to the product price this higher execution time is to be expected as multiple conditional assumptions are involved with some of them being rejected by the negotiating parties.

5.2.3 Experiment: PolicyGen

Early jNQS experiments using the PolicyGen PBs highlighted a number of issues surrounding experiment design and negotiation frameworks. Initially, the agent and opponent used their own instances the same PB to negotiate over a randomly selected authorisation rule from this PB. Assumptions were generated by backtracking this rule's body and grounding each predicate with respect to either the agent or opponent. The problem with this approach is that despite having different assumptions the agent and opponent end up agreeing with each other. In turn, negotiations are resolved in 1 or 2 rounds. To address this the use of different PolicyGen PBs for the agent and opponent was considered. However, during initial testing new problems arose. Given the way PolicyGen produces PBs it is extremely difficult to ensure each PB can "mesh" together to produce experimentally useful negotiations.

Experiments in this section are based on a compromise between the above two approaches; PB used in the A+B and S+S experiments are extended using PolicyGen PBs. For instance, the agent and opponent PBs used for experiment A+B 1 has the rules of a PolicyGen PB added to it. This allows for the negotiation outcomes to remain predicable, while increasing the time needed to compute the answer sets at each round. As the rule extension is simply to increase the compute time the PB of the agent and opponent are extended using the same set of rules. Results are recorded similarly to previous experiments, except given the size of some of the PBs involved an average it not taken.

S+S 1 Extension

To test performance of negotiations which conclude successfully (grant access to the requester) we extend experiment S+S 1 from Table 5.1. S+S 1 was selected as it took the longest of the experiments shown in the table. The query and assumptions from S+S 1 are retained. Table 5.2 summarise these extended experiments. Column **Experiments** contains the name of each experiment. This name is a composite of S+S 1 and the name of the PolicyGen PB extending it. For example, Gen03SS01 denotes S+S 1 extended by Gen03. The time for each negotiation to reach a conclusion is reported in column **Time** and **Rounds** . Since all of the experiments result in the requester being granted access the conclusion is not included in Table 5.2.

Recalling Table 3.4 shows the S+S 1 Extension experiments consider a range of PB sizes. Experiments Gen01SS01, 02, 03, and 04, cover agents and opponents with a "small" PB of around 20 to 90 ASP rules. "Normal" or "average" policy sizes, 140 to 600 rules, are covered by 05, 06, 07, and 08. Gen09SS01, 10, and 11

Experiment	Time (s)	Rounds
Gen01SS01	0.44	6
Gen02SS01	0.62	8
Gen03SS01	0.83	6
Gen04SS01	0.90	9
Gen05SS01	1.04	8
Gen06SS01	1.42	8
Gen07SS01	3.73	8
Gen08SS01	6.64	9
Gen09SS01	9.14	8
Gen10SS01	13.06	8
Gen11SS01	21.22	6
Gen12SS01	94.88	8
Gen13SS01	392.84	8
Gen14SS01	585.72	8
Gen15SS01	1030.49	8
Gen16SS01	1186.79	8
Gen17SS01	3828.00	6
Gen18SS01	17253.60	10

Table 5.2: PolicyGen Experiment, S+S Extension.

with a range of 1000 to 2000 rules denote the upper limit of what we consider a real world PB would contain. This leaves experiments 12 through 18 with PBs containing 4000 to 57000 rules. These policies are so impractically large they only serve to highlight possible optimisations for jNQS. As both the agent and opponent have their own PB each experiment negotiates over double the number of rules just outlined and shown in Table 3.4.

As it can be seen in Table 5.2 small PBs all grant the opponent access in less than a second. Normal size PBs begin to take a noticeable, but still tolerable, amount of time to conclude with Gen08SS01 taking around 7 seconds. Experiment Gen11SS01 with nearly 2000 rules takes a slow 22 seconds to conclude. The large PB considered by Gen12SS01 take over a minute to negotiate, while Gen18SS01 takes an impractically long 5 hours to conclude.

From these results it can be seen that jNQS has reasonable performance for PBs up to 1000 rules. On the other hand, negotiations over policies exceeding 2000 rules, as the case in experiments Gen12SS01 to Gen18SS01, show the need for further optimisation. Notably the number of rounds taken to complete a negotiation did not change significantly as the PB size increases. This suggests jNQS's performance is related to time needed to compute answer sets.

Each round the agent/opponent generates new answer sets using their PB, assumptions, and the offer presented by their opponent. However, this approach can result in time being wasted computing answer sets which have not changed from the previous round. This issue could be addressed by having the agent store answer sets between rounds and only computing new ones if the conditional proposal warrants it.

Another consideration is the choice of ASP solver used by jNQS. Currently it utilises DLV through the DLVWrapper API. This was a choice based largely on the convenience of the API, rather than the performance of DLV. As shown by Dodaro et al. [31] DLV is outperformed by well established solvers such as ClaspD [41], and even a preliminary prototype of the solver WASP [31]. Though the input syntax of ASP is fairly universal each implementation has its own nuances. For instance DLV introduces non-standard syntax to implement specific features utilised in Chapter 6, while SocACL's semantics take into account DLV's aggregate implementation. Furthermore, our system implementations presented throughout this thesis are all reliant on the DLVWrapper API, making any transition to a different, higher performance solver non-trivial. Given WASP uses DLV as it grounder it would be interesting to explore WASP's support for many of DLV's features. As such the optimisation of jNQS is left as a consideration for future development.

A+B 3 Extension

Here experiments extend A+B 3 from Table 5.1 to consider negotiations where the requester is not granted access. As with the S+S 1 extension, A+B 3 was selected since it took the longest of the experiments presented in Table 5.1. Furthermore, the query and assumptions are retained from A+B 3. Table 5.3 summarises the results of extending A+B 3.

As with the results shown Table 5.1 it takes significantly less time to determine unsuccessful access requests than successful. Since the results found in Table 5.3 are consistent with our earlier comments on unsuccessful negotiations and discussion of jNQS's performance concerns no further analysis is given.

5.3 Chapter Summary

This chapter introduced and evaluated jNQS, a implementation of the negotiation based query formalised in Chapter 4. We began with an overview of jNQS's Java classes in Section 5.1.1. The chapter concluded with Section 5.2 where we present the results of various experiments evaluating the performance of jNQS.

Experi.	**Time (s)**	**Rounds**
Gen01AB03	0.22	3
Gen02AB03	0.25	3
Gen03AB03	0.40	3
Gen04AB03	0.28	3
Gen05AB03	0.39	3
Gen06AB03	0.60	3
Gen07AB03	0.90	3
Gen08AB03	1.19	3
Gen09AB03	1.82	3
Gen10AB03	2.91	3
Gen11AB03	4.12	3
Gen12AB03	10.62	3
Gen13AB03	40.51	3
Gen14AB03	59.33	3
Gen15AB03	113.70	3
Gen16AB03	132.26	3
Gen17AB03	536.02	3
Gen18AB03	1502.38	3

Table 5.3: PolicyGen Experiment, A+B Extension.

Chapter 6

Attribute-Based Access Control Policy Update

6.1 Introduction

Previously discussed in Chapter 1, OSNs attempt to model the offline social structures of their users. As a user meets new people, changes jobs, etc., these structures change. This in turn impacts on the correctness of a user's privacy settings, potentially leading to unwanted or undesirable access control outcomes. One approach to addressing this issue is to update the user's PB with respect to this change. To illustrate this lets us consider the following rules from Alice's SocACL PB:

```
alice says alice.allow.A.view."cats.jpg".social if A.memberOf."UoL
  Lacrosse", not A.memberOf."UoL Tennis", A != alice;
alice says alice.allow.A.view."dogs.jpg".social if A.memberOf."UoL
  Lacrosse", not A.memberOf."UoL Coffee Lovers", A != alice;
alice says alice.memberOf."UoL Lacrosse";
```

Suppose Alice leaves the Lacrosse club on bad terms to join the recently formed Hockey club. This scenario conflicts with Alice's above rules as they:

- Claim Alice is a member of a club she just left.
- Do not consider her new club membership.
- Results in undesirable access control outcomes by granting view permissions to Lacrosse club members.

Clearly Alice is faced with a situation where it would be advantageous to update her PB. As discussed in Section 1.2.3, there is a lack of research on the update of ABAC policies. On the other hand, the update of ASP and other logic programs has been widely researched [32, 38, 73]. By leveraging the ASP semantics of SocACL it is possible to adapt logic programming techniques to the problem of ABAC policy update.

This chapter presents such an approach to the update of ABAC policies by adapting the work of Sakama et al. [73]. We being with an overview of the framework itself. This is followed by an analysis of the framework's semantic properties. The following chapter, Chapter 7, introduces and outlines jUpABAC, a prototype implementation of our update approach.

6.2 Policy Base

Similar to Chapter 4, PBs are encoded as ASP programs rather than as SocACL PBs. This is again done to ensure the generality of our update approach and to avoid the notational overhead of the SocACL to ASP translation. For this reason the update of SocACL PBs is postponed until Section 7.3.2. The formal representation of ASP PBs has been adapted from NKBs found in Chapter 4 with a two key changes. First, negative assumptions have been removed. Second, to avoid potential confusion this adaptation has also been renamed to simply PBs.

Definition 6.1 (*Policy Base*)
Policy Base (PB) is a triplet $\langle \Pi, H^+ \rangle$ where:

- Π is a set of rules of form (1.1).
- H^+ is the set of atoms called *Positive Assumptions* which the agent safely assumes to be true such that $H^+ \cap Head(\Pi) = \emptyset$ and $H^+ \cap H^- = \emptyset$.

In the above, Π is an ASP program representing a principals' ABAC PB. H^+ is the set of *positive assumptions*. Recalling Definition 4.1, NKBs have both a set positive and negative assumptions. As negative assumptions are not used by the policy update they have been removed.

The purpose of negative assumptions is to ensure the agent considered NAF decision criteria during negotiation. For our update this is unnecessary. As elaborated upon later, the update presented in this chapter is reactive. An update is initiated upon an observed, but unwanted access control outcome. Since the agent updating their PB has observed their opponent acting on an authorisation, the agent can safely assume the opponent does not violate the rule. Hence, they do not need to consider the negative assumptions. On the other hand, they do need to consider positive assumptions as they are used establish which rules in the agent's PB are causing the unwanted access control outcome.

Example 6.1 ***(Dan's PB)***

Dan is responsible for administering a large online photo gallery used by all UoL sporting clubs. He authors the PB $P = \langle \Pi, H^{+} \rangle$ for this gallery. Π contains the rules:

```
1 allow(dan, A, "write", "UoL Sports Gallery", social) :- memberOf(A, A,
    "UoL Sports").
2 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse").
3 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis").
4 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey").
5 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hotdog Eating").
```

Line 1 is an authorisation rule granting members of UoL sports clubs write permissions to the Sports Gallery. This allows members of each club to add new photos to the gallery themselves, instead of having Dan do it for them. Lines 2 to 5 results in the Lacrosse, Tennis, Hockey, and Hotdog Eating clubs being considered sports clubs, in turn allowing their members to add photos to the gallery. For the purposes of updates Dan assumes everyone is a member of all the clubs. Such that, $H^{+} =$

```
memberOf(carl, carl, "UoL Lacrosse")
memberOf(carl, carl, "UoL Tennis")
memberOf(carl, carl, "UoL Hockey")
memberOf(carl, carl, "UoL Hotdog Eating")
```

Note in the above, for brevity, only shows instances of Carl. For the other users in our running case study simply substitute the identifier for Carl with that of Alice, Bob, Dan, or Ellen.

6.3 Update Request

A Update Request (UR) encodes the information with which the PB is changed with respect to. Depending on the update approach this information varies. In the ASP update methodology developed by Eiter, et al. [32] a UR is a set of rules. Rules are never removed from the updated program. Instead their method combines the “new” rules with the “old” rules, while resolving rule conflicts by having “new” rules take precedence over “old” rules when solving the program.

For our approach a PB update is viewed as reactionary. Updates are initiated in response to an observed, but unwanted or unintended access control outcome. For example, members of the Lacrosse team can still view Alice's "cats.jpg" picture. This approach leads to a "deletion centric" solution where the rule/s causing these unwanted outcomes are removed to stop them occurring. In addition to removing rules an update also needs to be able to include new ones. To accommodate both these needs a UR takes the form of the following pair:

Definition 6.2 (*Update Request*)
For a PB $P = \langle \Pi, H^+ \rangle$ a *Update Request* is a pair $\langle O, U \rangle$ where:

- O is the set of grounded atoms, such that $\Pi \cup H \models O$ or $O = \emptyset$, and;
- U is the set of rules of the form (1.1).

O is a set of unwanted policy outcomes, while U is a set of new rules to be added to the updated PB.

Example 6.2 (*Dan's UR*)
At some point after deploying the PB from Example 6.1 Dan performs routine maintenance on the system. While doing this he notices a number of non-sports related photos being uploaded to the gallery by Carl. Dan has also been informed of the formation of a new UoL Swimming club which also needs access to the gallery. To perform an update that achieves both of these changes Dan defines a UR $\mathcal{UR} = \langle O, U \rangle$ where O contains:

```
allow(dan, carl , "write", "UoL Sports Gallery", social)
```

In this instance Dan has observed unwanted access while maintaining the system he oversees. Since Dan notices Carl is uploading these unwanted photos Dan adds the above authorisation to O. U contains:

```
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming").
```

To allow the new Swimming club to contribute to the sports gallery Dan adds a new rule. This rule, shown above, is similar to the rules for the other UoL sporting clubs by inferring the Swimming club is a sports club.

6.3.1 Characterising Updates

Central to the design of the update presented in this chapter is the intended result of applying a UR to a PB. Formally we consider this result as a characterisation of the update of a PB $P = \langle \Pi, H^+ \rangle$ w.r.t. a UR $\mathcal{UR} = \langle O, U \rangle$ such that:

$$(\Pi \backslash \Delta) \cup H \cup U \not\models O, \; \textit{where} \; \Delta \subseteq \Pi \tag{6.1}$$

Updates remove a set of rules Δ from Π which cause $\Pi \cup H^+ \cup U \models O$ to hold. Rules forming the original PB in Π are considered expendable, while new rules in U *must* be included in the updated PB.

This guaranteed inclusion makes the adding of U to the updated PB trivial as it can be achieved using a simple union operation. On the other hand, finding a set of rules Δ which can be removed from Π such that the above characterisation holds is non-trivial. As such, the focus of our update approach is finding this set Δ.

6.4 Update Program

The problem of finding Δ is treated as a hard search problem, a category of problems well suited to ASP. Using ASP solvers it is possible to test different subsets of rules from Π to find ones that do not infer O. This can be done using two different approaches. The first is to test each program $\Pi_i \in 2^{\Pi}$, where 2^{Π} is the powerset of Π, to find subsets of Π which do not infer O. Alternatively, and the approach we use, it is possible to define a derivative of Π, called an Update Program (UP) and denoted by Π', where the rules:

- Can be toggled "on" or "off" dynamically to influence its answer sets; and
- These answer sets report the toggle status of the program's rules.

Answer sets of the UP correspond to rule combinations where O does not hold. Since the goal of Δ is to contain rules which can be removed from Π to stop the inference of O, it follows that Δ is derived from these answer sets by populating it with rules that were toggled "off". However, answer sets do not report the rules which generated it, only information inferred from the program.

To circumvent this trait of answer sets, UPs make use of a common ASP technique; rule naming. By naming each rule in Π' it is possible for its answer sets to contain the names of the rules which inferred it. From these names it is straightforward to derive the rules themselves. Rule naming is performed in our update using the following bijective function.

Definition 6.3 (***Rule Naming***)

For a set R of rules of the form (1.1) and a set N of names the bijective function $\mathcal{N} : R \rightarrow N$ maps a rule to a name. Conversely, $\mathcal{N}^{-1} : N \rightarrow R$ denotes the reverse; names mapped to rules.

Rule names serve a dual purpose in UPs. First, they provide an identifier by which the rules from Π are toggled. Second, the rule names appear in the UPs answer sets to report the toggle status of each rule that produced the answer set. One problem with dynamically toggling rules to find combinations is that not all of these combinations are useful. Consider an answer set of Π' where all rules are toggled off. Clearly this rule combination cannot derive O because there are no rules "on" which could infer it. Furthermore, this answer set denotes a Δ which removes all rules in Π, which is obviously an absurd update if other options exist. To avoid these "useless" rule combinations UPs utilises a DLV feature called *weak constraints* [33, 20] to preference certain rule combinations.

Definition 6.4 (***Weak Constraints***)

Any program P consisting of rules of the form (1.1) can also contain rules of the form:

$$\underset{w}{\leftarrow} L_1, \cdots, L_m, not\ L_{m+1}, \cdots, not\ L_n. \tag{6.2}$$

Called a *weak constraint*, where each $L_i, 1 \leq i \leq n$ is a literal. Such that the answer sets $Ans(P)$ are limited to those where the number of unsatisfied weak constraints in P is minimal.

The semantics of these weak constraints, as explained by Buccafurri et al. [20] "...minimises the number of violated instances of [weak] constraints".

Example 6.3 (***Weak Constraints***)

For the programs $P = \{a \leftarrow\ not\ b.\ b \leftarrow\ not\ a.\}$, $W_1 = \{\leftarrow a.\ \leftarrow b\}$, and $W_2 = \{\underset{w}{\leftarrow} a.\ \underset{w}{\leftarrow} b\}$. P has two answer sets; $\{a.\}$ and $\{b.\}$. $P \cup W_1$ has no answer sets since both answer sets of P violate the constraints in W_1. On the other hand, $P \cup W_2$ has two answer sets; $\{a.\}$ and $\{b.\}$. This is because though both answer sets of P violate the weak constraints in W_2 the number of violated instances is minimal, i.e. each answer set of $P \cup W_2$ violates one weak constraint.

As illustrated in the above example, weak constraints restrict a program's answer sets to those which violate as few weak constraints as possible. Weak constraints are used in UPs to minimise the number of rules toggled "off". This results in the update favouring rule combinations where fewer rules are removed. With this in mind, UPs are defined as follows:

Definition 6.5 (***Update Program***)
For a PB $\langle \Pi, H^+ \rangle$ and a UR $\langle O, U \rangle$ there is an *Update Program* (UP) Π', such that Π' contains:

- For every atom $o \in O$, the constraint:

$$\leftarrow o. \tag{6.3}$$

- For every rule $(\Sigma_i \leftarrow \Gamma_i) \in \Pi$ with the name $\gamma_i = \mathcal{N}(\Sigma_i \leftarrow \Gamma_i)$ and its "inverse-name" $\bar{\gamma}_i$, the rules:

$$\Sigma_i \leftarrow \Gamma_i, \gamma_i. \tag{6.4}$$
$$\gamma_i \leftarrow \text{not } \bar{\gamma}_i. \tag{6.5}$$
$$\bar{\gamma}_i \leftarrow \text{not } \gamma_i. \tag{6.6}$$
$$\underset{w}{\leftarrow} \textit{not } \gamma_i. \tag{6.7}$$

- For every rule $(\Sigma_j \leftarrow \Gamma_j) \in U$, the rule:

$$\Sigma_j \leftarrow \Gamma_j. \tag{6.8}$$

Equation (6.3) is a integrity constraint which ensures answer sets of the UP do not infer the unwanted policy outcomes O. Equation (6.4) is a rule from Π with its unique name appended to the rule body. This causes (6.4) to become active or inactive (toggled on or off) depending on (6.5) and (6.6), where γ_i denotes toggled on and $\bar{\gamma}_i$ toggled off.

Definition 6.6 (***Active/Inactive Rules***)
Given a PB $\langle \Pi, H^+ \rangle$, UR $\langle O, U \rangle$, their UP Π' and a some model M of Π'. We say that the rule $(\Sigma_i \leftarrow \Gamma_i, \gamma_i.) \in \Pi'$ is *active* w.r.t. M if $\gamma_i \in M$. Otherwise, it is *inactive*.

Similar to the small example program in Section (1.2.3), (6.5) and (6.6) yields answer sets which contain either γ_i or $\bar{\gamma}_i$, but not both. As a result in some answer sets of UP rules of the form (6.4) will be active, while in others it will not. (6.7) is a weak constraint [79] to ensure the answer sets of Π' have a maximal number of (6.4) rules active. Finally, (6.8) includes the new rules from U in the UP so the new rules are considered while testing rule combinations.

Example 6.4 (***Update Program***)

For the PB P and UR $\mathcal{UR}$ from Example 6.1 their UP Π' contains the following:

```
 1 :- allow(dan, carl, "write", "UoL Sports Gallery", social).
 2 allow(dan, A, "write", "UoL Sports Gallery") :- memberOf(A, A, "UoL
     Sports"), r0.
 3 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse"), r1.
 4 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis"), r2.
 5 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hotdog Eating"),r3.
 6 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey"), r4.
 7 r0 :- not neg_r0. r1 :- not neg_r1. r2 :- not neg_r2.
 8 r3 :- not neg_r3. r4 :- not neg_r4. neg_r0 :- not r0.
 9 neg_r1 :- not r1. neg_r2 :- not r2. neg_r3 :- not r3,
10 neg_r4 :- not r4.
11 :~ not r0. :~ not r1. :~ not r2. :~ not r3. :~ not r4.
12 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming" ).
```

Line 1 is the integrity constraint derived from O using (6.3). Lines 2 to 6 are the rules of Π with the names appended to the rule body (6.4). These rules are made active or inactive by lines 7 to 10 (6.5) (6.6). Line 11 are the weak constraints for lines 7 to 10. New rules from U are included on line 12 (6.8).

6.5 Update Candidate

Answer sets of the UP contain the names of rules used to infer it. These names are then used to derive the rules themselves which in turn form Δ. However, this presents a problem. Inherent to the stable model semantics [42] on which ASP is based programs may have multiple answer sets. This means for a given UR and its subsequent UP there are multiple sets of rules which could be removed from Π such that O does not hold and the number of active rules has been maximised. As all of these sets represent reasonable options for Δ they are referred to as Update Candidate (UC)s.

Definition 6.7 (*Update Candidate*)

For a PB $\langle \Pi, H^+ \rangle$ and a UR $\langle O, U \rangle$ let Π' be their UP such that for the answer sets:

$$M_i \in Ans(\Pi' \cup H^+) \tag{6.9}$$

There is a set of rules $C_i = \mathcal{N}^{-1}(\mathcal{N}(\Pi') \backslash M_i)$ called an *Update Candidate* (UC). The set of all update candidates for a PB P and UR $\mathcal{UR}$ is denoted by $\mathcal{UC}(P,\mathcal{UR})$.

M_i contains the names of all active rules which produced the answer set. $\mathcal{N}(\Pi')$ contains the names of all rules in Π'. $\mathcal{N}(\Pi') \backslash M_i$ is the set of all rules in Π', excluding the names of active rules in M_i, leaving only the names of inactive rules. $\mathcal{N}^{-1}(\mathcal{N}(\Pi') \backslash M_i)$ contains the inactive rules.

Example 6.5 (*Update Candidate*)

For the UP Π' along with the PB P and UR $\mathcal{UR}$ from Example 6.4, $Ans(\Pi' \cup H^+)$ contains two models; M_1 and M_2. M_1 contains:

```
neg_r3, r0, r1, r2, memberOf(carl, carl, "UoL Hotdog Eating"), r4
```

We see M_1 reports rules `r0`, `r1`, `r2`, and `r4` are active, while `neg_r3` denotes rule `r3` is inactive. On the other hand, M_2 contains:

```
memberOf(carl, carl, "UoL Sports"), r1, r2, r3, memberOf(carl, carl, "
  UoL Hotdog Eating"), neg_r0, r4
```

In M_2 `r1`, `r2`, `r3`, and `r4` are active rules, while `r0` is inactive. $\mathcal{UC}(P,\mathcal{UR})$ contains two update candidates; C_1 and C_2. C_1 corresponds to M_1 and contains:

```
{memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hotdog Eating").}
```

While C_2 corresponds to M_2 and contains:

```
{allow(dan, A, "write", "UoL Sports Gallery", social) :- memberOf(A, A,
   "UoL Sports").}
```

In Example 6.5 we are presented with two update candidates which support the running example's UR. Though the removal of either C_1 or C_2 fulfils the UR, C_1 or C_2 clearly have different levels of impact on the updated policy. If C_2 is applied to the PB none the other sports clubs would be able to write to the gallery since it removes the authorisation. On the other hand, applying C_1 only prevents the Hotdog Eating club from performing writes by stopping it being considered a sports club. Intuitively, C_1 "better" since it has fewer "side effects" when compared to C_2.

As noted in the previous section, some UCs are "better" than others. Some remove fewer rules, while some may remove less "important" rules. Intuitively, Δ should represent the best set of rules to remove. Since a concept of "best" is rather subjective the framework attempts to quantify a measure of "best-ness" based on the *principle of minimal impact.* For the update of any PB w.r.t a UR, if there are multiple UCs, then the one that causes the least change or lowest impact should be applied.

This measure differs from the objective of the UP's weak constraints. Weak constraints minimise the number of rules removed regardless of the rule's importance in the PB. On the other hand, our proposed metric determines which combinations of rules would cause the least impact on the outcomes of the PB if they were removed. To quantify a UC's impact on the PB a heuristic called the *Update Impact Value* (UIV) is used.

Definition 6.8 (*Update Impact Value*)

For a PB $P = \langle \Pi, H^+ \rangle$, the UR $\mathcal{UR} = \langle O, U \rangle$ and their update candidates $\mathcal{UC}(P,\mathcal{UR})$, each UC $UC_i \in \mathcal{UC}(P,\mathcal{UR})$ has an Update Impact Value (UIV):

$$UIV(UC_i, \Pi) = \sum_{r_j \in \Pi} |H(r_j) \cap B(UC_i)| + |B(r_j) \cap H(UC_i)| \tag{6.10}$$

As the name suggests, a UC's Update Impact Value (UIV) is a value representing how much change a UC would cause to the given PB. This is done by finding how many rules in Π infer or are inferred by atoms in a given UC. $|H(r_j) \cap B(UC_i)|$ is the cardinality of a set of atoms in both the Head of rules in Π and in the Body of rules of the UC. $|B(r_j) \cap H(UC_i)|$ is the cardinality of the set of atoms in both the Body of rules in Π and in the Head of the UC's rules.

Example 6.6 (*Update Impact Value*)

Considering the PB and UR from Examples 6.1 and 6.2 along with their UCs C_1 and C_2 from Example 6.5. $UIV(P, C_1) = 1$ while $UIV(P, C_2) = 4$.

Calculating the UIV of these two UCs yields results in line with our comments on Example 6.5. The "best" UC is now selected based on the UIV.

Definition 6.9 (***Update Candidate*** Δ)

For a PB $P = \langle \Pi, H^{+} \rangle$, a UR $\mathcal{UR}$, and their UCs $\mathcal{C} = \mathcal{UC}(P,\mathcal{UR})$ there is a UC $\Delta \in \mathcal{C}$ such that there exists no other UC $C \in \mathcal{C}$ where $UIV(C, P) < UIV(\Delta, P)$.

In other words, from set of UCs the one with the lowest UIV is selected to be Δ. In the case of Example 6.6, $\Delta = C_1$. With Δ selected it is now applied in the update of PB.

6.6 Applying the Update

With Δ now selected, let s again consider the characterisation of updating a PB by a UR shown in Equation (6.1). In the equation new rules from U are added to the existing rules in Π, while the rules which populate Δ are removed. As such the update operator $\oplus$, which is distinct from $\oplus$ in Chapter 4, is defined as follows:

Definition 6.10 (***Policy Update***)

For a PB $P = \langle \Pi, H^{+} \rangle$ and the UR $\mathcal{UR} = \langle O, U \rangle$, the update of P w.r.t. $\mathcal{UR}$ is the PB:

$$\begin{aligned} P \oplus \mathcal{UR} =& \langle \Pi \backslash \Delta \cup U, H^{+} \rangle & (6.11) \\ =& \langle \Pi^{*}, H^{+} \rangle & (6.12) \\ =& P^{*} & (6.13) \end{aligned}$$

Where the UC $\Delta \in \mathcal{UC}(P,\mathcal{UR})$ and has a minimal UIV, as per Definition 6.9.

The operation $P \oplus \mathcal{UR}$ yields an updated PB; P^{*}. $P^{*} = \langle \Pi^{*}, H^{+} \rangle$ such that $\Pi^{*} \cup H^{+} \not\models O$. The PB also contains all the rules in U, so $U \subseteq \Pi^{*}$.

Example 6.7 (***Applying*** Δ ***to Dan's PB***)

For the PB P from Example 6.1 and the UR $\mathcal{UR}$ from Example 6.2, $P \oplus \mathcal{UR} = \langle \Pi^{*}, H^{+} \rangle$. Π^{*} contains the rules:

```
allow(dan, A, "write", "UoL Sports Gallery", social) :- memberOf(A, A,
  "UoL Sports").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse").
```

```
3 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis").
4 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey").
5 memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming").
```

Lines 1 to 4 have been retained from Π while the contents of Δ, the rule for the Hotdog Eating club, has been removed. Line 5 shows the new rule from U which allows the Swimming club to upload to the sports gallery by making it be considered a sports club.

6.7 Semantic Properties

This section investigates semantic properties of the update operator $\oplus$ introduced in Section 6.6. Here the ability for $\oplus$ to support common update scenarios is explored.

6.7.1 Initialising a PB

So far this thesis has focused on updates to PB which have already been populated with rules. Though this reflects the most common state of a PB it ignores a critical period of the PB's existence; its initialisation. In the context of this chapter's update formalism, an *initialisation* is defined as a property of a UR as follows:

Property 6.1 (*Initialisation*)

For a PB $P = \langle \emptyset, H^+ \rangle$ a UR $\mathcal{UR} = \langle O, U \rangle$ is called a *Initialisation* of P, such that:

$$\begin{aligned} P \oplus \mathcal{UR} &= \langle \emptyset \backslash \Delta \cup U, H^+ \rangle && (6.14) \\ &= \langle U, H^+ \rangle && (6.15) \end{aligned}$$

An initialisation is where a PB containing no rules is updated to contain rules. In the above $\emptyset \backslash \Delta \cup U = U$ since $\emptyset \backslash \Delta = \emptyset$, therefore $\oplus$ supports the initialisation of a PB.

6.7.2 Update Types

Due to the lack of research on ABAC policy update the functionality of $\oplus$ is evaluated by considering useful properties from the fields of knowledge and logic program update. In their work on knowledge update Alchourrón et al. [3] consider updates in terms of three operations; *expansion*, *contraction*, and *revision.* Expansion is adding new information to the updated knowledge, contraction is the removal, while revision does both. Taking these high-level explanations these operations are formally defined in the context of our update.

Definition 6.11 (***Update Operations***)
Let $P = \langle \Pi, H^+ \rangle$ be a PB, $\mathcal{UR} = \langle O, U \rangle$ be a UR and $P \oplus \mathcal{UR} = \langle \Pi^*, H^+ \rangle = P^*$ be P updated by $\mathcal{UR}$. We say that P^* is the result of an:

- *Expansion* when $\Pi \subset \Pi^*$.
- *Contraction* when $\Pi^* \subset \Pi$.
- *Revision* when $\Pi \neq \Pi^*$.

An expansion is an update where the updated PB is a superset of the original. Conversely, a contraction is when the original PB is a superset of the updated PB. If the update is neither an expansion nor contraction, and results in the updated PB that is different to the original then it is a revision.

In [3] each of these operations is defined over its own operator. On the other hand, our update is defined over the single operator $\oplus$. This superficially suggests that $\oplus$ supports one update operation; revision. Let us consider properties of a UR, such that specially formed URs can capture the update operations in Definition 6.11

Property 6.2 (***UR Type***)
For a PB $\langle \Pi, H^+ \rangle$ and a UR $\langle O, U \rangle$ we classify the UR as follows:

1. If $O = \emptyset$ and $U \neq \emptyset$ then the UR is an *expansion.*
2. If $O \neq \emptyset$ and $U = \emptyset$ then the UR is an *contraction.*
3. If $O \neq \emptyset$ and $U \neq \emptyset$ then the UR is an *revision.*

Proof 6.1 Let us consider the PB $P = \langle \Pi, H^+ \rangle$ and the UR $\mathcal{UR} = \langle O, U \rangle$. As per Definition 6.11 the results of updating P by $\mathcal{UR}$ is an expansion if $P \oplus \mathcal{UR} = \langle \Pi \backslash \Delta \cup U, H^+ \rangle$, where $\Pi \subset \Pi \backslash \Delta \cup U$. For $\mathcal{UR}$ to be an expansion as per Property 6.2 $O = \emptyset$ and $U \neq \emptyset$, such that Definition 6.11 holds.

Let Π' be the UP of P and $\mathcal{UR}$. When $O = \emptyset$, Π' must contain no constraints of the form (6.3). As a consequence all models $M_i \in Ans(\Pi' \cup H^+)$ are only constrained by the weak constraints (6.7) which maximise the number of active rules. Since nothing in Π' is preventing the weakly constrained rules from being active in every M_i contains all the rules names from $\mathcal{N}(\Pi)$. As a result $\mathcal{UC}(P,\mathcal{UR}) = \emptyset$, and in turn $\Delta = \emptyset$. When $\Delta = \emptyset$, $\Pi \subset \Pi \backslash \emptyset \cup U = \Pi \subset \Pi \backslash \Delta \cup U$, which is clearly a expansion under Definition 6.11. Therefore the UR $\mathcal{UR} = \langle \emptyset, U \rangle$ encodes an expansion of P.

Lets again consider the PB $P = \langle \Pi, H^+ \rangle$ and the UR $\mathcal{UR} = \langle O, U \rangle$. By Definition 6.11 the update of P by $\mathcal{UR}$ is a contraction if $P \oplus \mathcal{UR} = \langle \Pi \backslash \Delta \cup U, H^+ \rangle$, where $\Pi \backslash \Delta \cup U \subseteq \Pi$. For $\mathcal{UR}$ to be a contraction as per Property 6.2 $O \neq \emptyset$ and $U = \emptyset$, such that Definition 6.11 holds. By substituting $U = \emptyset$ into $\Pi \backslash \Delta \cup U \subset \Pi$ yields $\Pi \backslash \Delta \subset \Pi$ which clearly holds as a contraction under Definition 6.11. Therefore UR $\mathcal{UR} = \langle O, \emptyset \rangle$ encode a contraction of P.

By Definition 6.10 a revision UR trivially holds, so the proof is omitted.

□

PBs which contain no rules can be initialised by a UR with a nonempty U-set. In addition to this, by ensuring a given UR adheres to Property 6.2 major update operations of expansion, contract, and revision as define under Definition 6.11 can be performed through using our $\oplus$ operator.

6.8 Chapter Summary

In this chapter we have presented a formal approach to updating ABAC policy bases. We began in Section 6.1 by demonstrating the need for updates in our case study. In Section 6.2 we introduced a formal representation for PBs derived from the one used in Chapter 4. Following this, in Section 6.3 we introduced and discussed the formal representation of the updates themselves; URs. These URs are then used in conjunction with the PB to be updated to form a UP, which we presented in Section 6.4.

The UP is used to compute UCs, Section 6.5; combinations of rules to remove from the PB. In Section 6.6 we formalised how the UC is applied to the PB, along with the new rules defined in the UR. Concluding in Section 6.7, we provided an analysis and discussion of our update's semantic properties.

Chapter 7

Implementation of Policy Update

7.1 Introduction

As with Chapters 2 and 4, a prototype implementation of our update framework presented in Chapter 6 has been developed. This prototype, called jUpABAC, is written in Java and utilises DLVWrapper v4.2 to interact with DLV.

7.1.1 Technical Details

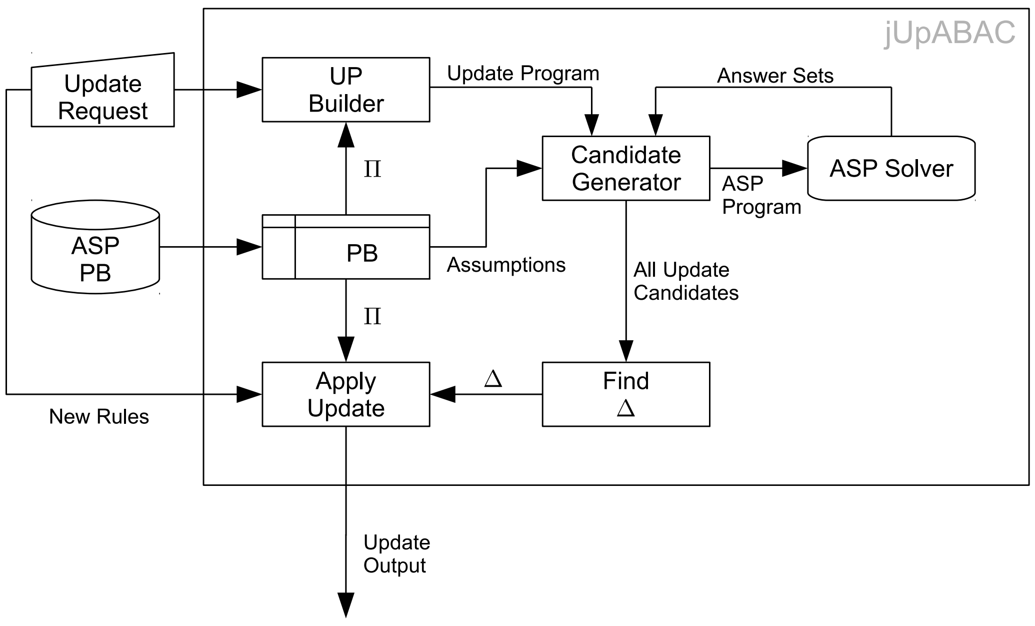


Figure 7.1: Update Prototype Flowchart.

Figure 7.1 summarises the interaction between jUpABAC's various modules. jUpABAC takes a PB encoded as an ASP program, a set assumptions, and a UR as input. The PB is stored along with the UR to build the UP.

The PB is used by the UP Builder module to generate a UP. This program and the PBs assumptions are used as input for the Candidate Generator. As the name suggests, this generator derives UCs from answer sets of the UP using DLV via DLVWrapper. Following this, the Find Δ module calculates the UIV for all UCs produced by the Candidate Generator to select one to be Δ. UC Δ is then applied along with the new rules from the UR to the ASP PB to produce the final output. Depending on the output options selected at runtime jUpABAC either outputs the complete updated PB, a list of the rules removed and added, or simply the rules in Δ.

7.2 Java Classes of jUpABAC

The modules illustrated in Figure 7.1 correspond to the following Java classes which make up jUpABAC. As most of these classes have standard "setter", "getter", toString, and display methods they are omitted for brevity.

Class: `Main`

The `Main` class coordinates all the module interactions illustrated in Figure 7.1. Taking a UR and PB as input, `Main` initialises the various class instances required for an update. The `Main` class has the following methods:

- `UpdateCandidate SelectDelta( Set<UpdateCandidate>, PolicyBase )`
- `int _UIV( UpdateCandidate, PolicyBase )`
- `PolicyBase applyDelta( PolicyBase, UpdateCandidate )`

`SelectDelta( )` implements Definition 6.9 to select Δ from the set of UCs. This method makes use of `_UIV( )` which calculates the UIV of each UC in the set w.r.t the PB as per Definition 6.8. Operator $\oplus$, Definition 6.10, is implemented by `applyDelta( )` and returns a updated version of the `PolicyBase`.

Class: `PolicyBase`

The `PolicyBase` class implements Definition 6.1. Since the definition is largely unchanged from NKB, `PolicyBase` is based on the same code as the `NKB` class from Section 5.1. `PolicyBase` has the following properties:

- `Set<String> Pi`
- `Set<String> APlus`

Note that since the update framework does not need negative assumptions they have been removed when adapting class `NKB`. As such, these `Set<String>`s correspond, respectively, to Π and H^+ in Definition 6.1.

Class: `UpdateRequest`

Definition 6.2 is implemented by the `UpdateRequest` class through the following properties:

- `Set<String> O`
- `Set<String> U`

Where these `Set<String>`s respectively denote the set O and U from Definition 6.2.

Class: `RxN`

Class `RxN` implements the rule to name mapping bijective function described by Definition 6.3. The class constructor takes a set of rules and a "name seed", a string on which to unique rule names are based. Names derived from the name seed are assign to each rule with the mapping stored in a `BidiMap`, a bidirectional hash map implementation from the Apache Commons package.

Class: `UpdateProgram`

The `UpdateProgram` class generates and stores the ASP program described by Definition 6.5 and has the following properties:

- `Set<String> PiPrime`
- `RxN N`

`PiPrime` is used to store the UP resulting from `_make( )`, while `N` is an instance of the `RxN` class. `N` is generated w.r.t. the input PB upon the invocation of `RxN`'s constructor. The various components of the UP are generated from the input PB and UR using the following methods:

- `String _appendRName( String Rule, String RName )`
- `String _RNameToggleRules( String RName )`
- `String _oConstraint( Stirng O )`
- `String _wconstForRName( String RName )`
- `Set<String> _make( PolicyBase PB, UpdateRequest UR )`

`_appendRName( )` appends the rule name `RName` to the body of `Rule`. The method `_RNameToggleRules( )` prepares the disjunctive rules that cause rules of the name `RName` to become active or inactive. Methods `_oConstraint( )` and `_wconstForRName( )` prepare an update program's collection of constraints and weak constraints from `Os` and `RNames`.

The above methods are utilised by `_make( )`, which passes the contents of a PB and UR to each of the above methods. Furthermore, `_make( )` combines the output of these various methods into a single `Set<String>`.

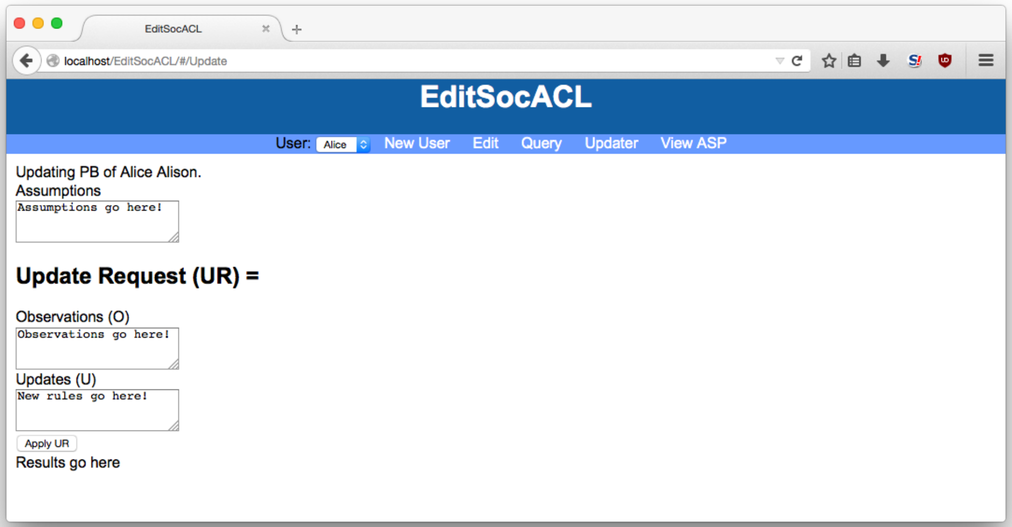
Class: `UpdateCandidate`

The `UpdateCandidate` class implements Definition 6.7 using the properties:

- `RxN N`
- `Set<Set<String>> UC`

Unlike `UpdateProgram`, `N` in `UpdateCandidate` is not generated. Here `N` is a copy of `N` from the `UpdateProgram` used in `UpdateCandidate`'s constructor. The computed UCs are stored in the set `UC`. The UCs are derived from answer sets computed by DLV which is accessed using methods from DLVWrapper in `UpdateCandidate`'s `Ans( )` method.

7.2.1 Updates using EditSocACL



Updating PB of Alice Alison.
Assumptions
```
Assumptions go here!
```
Update Request (UR) =

Observations (O)
```
Observations go here!
```
Updates (U)
```
New rules go here!
```
Apply UR
Results go here

Figure 7.2: Updating a PB.

Experiment	Π	**UR Type**	**UR** O	**UR** U	**UP**	**UC**
TVOff_01	4	Expansion	0	2	18	1
TVOff_02	5	Contraction	1	0	21	1
AliceHockey	25	Revision	3	1	116	2
DanAdmin	7	Revision	1	1	22	1

Table 7.1: Experiment Summary

Following the "Updater" link in the main menu of EditSocACL provides the form shown in Figure 7.2. The update process begins by first selecting a user using the "User" dropdown whose PB we wish to update. After this, one must provide the assumptions to be used in the update process. Once this has been done unwanted, but observed access control outcomes are listing in the appropriately label text area. Similarly, new rules to add to the selected user's PB are listed in its corresponding text area. Pressing the "Apply UR" button begins the update. The result of the update is displayed below "Results go here" and are "pushed" to the user's PB stored in EditSocACL's PB database.

7.3 Experiments

jUpABAC's performance along with the update formalism itself is evaluated using three experiment sets. To evaluate the formalism the first two experiment sets are based on the popular "TV Off" update example and the case study found in this chapter's introduction. These experiments have be run on a computer of the following specifications: Intel Core i7 2.9GHz, 8GB RAM MacBook Pro running OSX 10.10.3, Java RE 1.6.0_37, and DLV release (2012-7-12). The third experiment set uses the PBs generated using PolicyGen to evaluate jUpABAC's performance. Given the large size of some of these PB these experiments have been run on a computer of the following specifications: Intel Core i7 4.4GHz, 16GB RAM running Linux Mint 17.2 Rafaela, Java RE 1.6.0_37, and DLV release (2012-7-12).

Table 7.1 summaries the "TV Off" and chapter introduction experiments. Column Π is the number of rules the PB contains, while column **UR Type** is the UR type, as per Property 6.2, being applied to the PB. **UR** O and **UR** U shows the number of items in the sets O and U, respectively, of this UR. The number of UCs found for each experiment is reported in column **UC**. A summary of the PolicyGen experiments and results can be found in Table 7.3 using a similar column naming system as Table 7.1

For the experiments outlined in Table 7.1 the results are shown in Table 7.2. In each of these experiments the CPU time taken to perform an update is measured

Experiment	Avg. Time (ms)	Min Time (ms)	Max Time (ms)
TVOff_01	4.053	3	72
TVOff_02	4.036	3	62
AliceHockey	11.513	9	93
DanAdmin	7.642	6	76

Table 7.2: Experiment Performance Results

using the `Java.System.currentTimeMillis` method. It is called before and after a full update is performed. This includes all the classes initialisation, but excludes file IO. Each PB and UR combination is repeated 1000 times to provide an average. Results of the PolicyGen experiments have been recorded differently to account large size of some of the PB and subsequently longer update times. These differences will be discussed later in Section 7.3.3.

7.3.1 Experiment: TV Off

The "TV Off" scenario is a popular case study in the fields of knowledge and logic program update [32, 38, 73]. In this scenario a set of rules defines whether an agent watches TV or goes to sleep. With our update methodology based on logic program updates it is of interest to consider this classic case study. Let us begin by introducing the PB for TV Off.

Example 7.1 (***TV PB*** P)

Rules describing the decision logic forms part of a PB, such that $\Pi =$

```
1 sleep :- not tv_on.
2 watch_tv :- tv_on.
3 night. tv_on.
```

In the above, line 1 states the agent goes to sleep if the TV is not on, while on line 2 they watch TV if the TV is on. Line 3 contains two environmental attributes; its night time and the TV is currently turn on. These attributes remove the need for TV Off to consider assumptions as the agent already knows the state of the TV and time of day, thus $H^+ = \emptyset$.

The TV Off case study is often used to illustrate the logic of applying a sequence of updates such that the second update aims to "undo" the first. As shown in Table 7.2 this is achieved here by splitting the experiment into two parts; TVOff_01 and TVOff_02. Each experiment corresponds to the application of a different UR; $\mathcal{UR}_1$ for TVOff_01 and $\mathcal{UR}_2$ for TVOff_02.

Example 7.2 (***Power Failure*** $\mathcal{UR}$)

$\mathcal{UR}_1$ is an expansion such that, $O_1 = \emptyset$ and U_1 contains:

```
1 -tv_on :- power_failure.
2 power_failure.
```

These new rules encode a power failure. On line 1 the rule stats the TV will not be on if there is a power failure. Line 2 introduces the power failure as a environmental attribute. Note that these new rules conflict with those in P as the power failure causes the TV to not be on, while P states it is on.

$\mathcal{UR}_2$ is a contraction which attempts to undo $\mathcal{UR}_1$ by removing the power failure. For $\mathcal{UR}_2$, $O = \emptyset$ and U_2 contains:

```
-power_failure.
```

For this UR the set of new rules contains a single rule signifying an end to the power failure.

Let us now consider the update of P by $\mathcal{UR}_1$ to produce an updated PB P_1, which is then updated by $\mathcal{UR}_2$. For update $P \oplus \mathcal{UR}_1$ the UP of P and $\mathcal{UR}_1$ is $P_1' =$

```
night :- r0.
sleep :- not tv_on, r1.
watch_tv :- tv_on, r2.
tv_on :- r3.
r0 :- not neg_r0. r1 :- not neg_r1. r2 :- not neg_r2. r3 :- not neg_r3.
neg_r0 :- not r0. neg_r1 :- not r1. neg_r2 :- not r2. neg_r3 :- not r3.
:~ not r1. :~ not r0. :~ not r2. :~ not r3.
power_failure.
-tv_on :- power_failure.
```

Since $O_1 = \emptyset$, the UP P_1' contains no constraint for O_1. However, the update introduces a conflict where the TV is both on (`tv_on`) and off (`-tv_on`) at the same time. As a result, answer sets of the UP denote rule combinations where this conflict does not occur. Since `power_failure` and `-tv_on` are new rules they must be included. On the other hand, `tv_on` is an existing attribute, so it can be removed by the update. As such, the answer sets P_1' yield a single UC, shown as follows.

```
tv_on.
```

As this is the only UC it becomes Δ by default resulting in the updated PB P_1 containing the following rules:

```
night.
sleep :- not tv_on.
watch_tv :- tv_on.
power_failure.
-tv_on :- power_failure.
```

The above now allows the agent to conclude the TV is not on when there is a power failure. Furthermore, the environmental attributes have been updated with a `power_failure`, which caused `tv_on` to be removed.

Suppose the power failure has now ended and P_1 must again be updated to account for this. To do this let us perform the update $P_1 \oplus \mathcal{UR}_2$. The UP of this update is $P'_2 =$

```
night :- r0.
sleep :- not tv_on, r1.
watch_tv :- tv_on, r2.
power_failure :- r3.
-tv_on :- power_failure, r4.
r0 :- not neg_r0. r1 :- not neg_r1. r2 :- not neg_r2. r3 :- not neg_r3.
   r4 :- not neg_r4.
neg_r0 :- not r0. neg_r1 :- not r1. neg_r2 :- not r2. neg_r3 :- not r3.
   neg_r4 :- not r4.
:~ not r0. :~ not r1. :~ not r2. :~ not r3. :~ not r4.
-power_failure.
```

P'_2 also yields a single UC, show below.

```
power_failure.
```

Since it is the only UC it is Δ by default. Applying it to P_1 give the updated PB $P_2 =$

```
night.
-power_failure.
sleep :- not tv_on.
watch_tv :- tv_on.
-tv_on :- power_failure.
```

Intuitively, removing `power_failure` should cause the TV to be on again. However, this is not the case. This result highlights an interesting difference between our update and others. In our method the rules which allow O to hold are removed along with rules which contradict the new information/rules. As a side-effect, to "undo" an update the rules removed by the previous update need to re-added using U. This differs greatly from the update approach taken by Eiter et. al, [32]

In their update new rules are added in a way that overrides, but never removes, older ones. This causes the Eiter et. al, update not to exhibit the issue where the TV still being off after the power failure ends [32]. Arguably this is less intuitive as the power failure ending does not necessarily mean the TV is on again. Regardless, their [32] update focuses on the semantics of the updated program while our approach tackles both the semantics and syntax. From the performance results shown in Table 7.2 it can be seen there is little difference in average execution time between TVOff_01 and TVOff_02 with both values trending toward the minimal execution time. However, since these PBs are so small the results are of little interest.

7.3.2 Experiment: Alice Revision

Continuing the running case study of Alice this experiment considers the example from the chapter's introduction, Section 6.1. Here Alice's PB is updated w.r.t changes to her social circumstances. Let us begin by defining a UR which encodes Alice:

- Leaving the Lacrosse club on poor terms.
- Joining the Hockey club.
- Does not want Lacrosse club members viewing her photos.

Example 7.3 ($\mathcal{UR}_A$)

Alice notices Bob being granted access to her photos' `cat.jpg` and `dog.jpg` so she adds them to her set O_A along with her unwanted Lacrosse club membership. As such O_A =

```
allow( alice, bob, view, "cats.jpg", social )
allow( alice, bob, view, "dogs.jpg", social )
memberOf( alice, alice, "UoL Lacrosse" )
```

O_A contains two `allow` predicates resulting from Alice observing Bob accessing `cats.jpg` and `dogs.jpg`. To remove Alice's Lacrosse membership she treats it as an unwanted outcome. Alice adds her new Hockey club membership attribute by including it in the set U =.

```
memberOf( alice, alice, "UoL Hockey" )
```

The above UR is interesting as it has had to take into account nuances of SocACL's translations. Recall section 2.9. In SocACL authorisation conflicts, having both `allow` and `deny` at the same time, are resolved by the `action` predicate in UA. However, instead of observing the unwanted `action`, O_A targets the unwanted permission. This is done to avoid potential problems where the update removes UA rules from Alice's translated SocACL PB. Following this reasoning when updating SocACL PBs the translation does *not* contain the set UA.

To remove Alice's Lacrosse membership the experiment has taken liberties with the definition of URs. In the spirit of Definition 6.2, O_A contains predicates that denote observed, but unwanted access control outcomes. In this case $\mathcal{UR}_A$ has allowed her to include unwanted attribute. Let us now construct Alice's PB as per Definition 6.1.

Example 7.4 (***Alice's PB*** P_A)

For this section the experiment is being performed over Alice's full PB as found in Appendix A.2.1. Since this PB contains enough rules to fill several pages only a subset of the PB which is of interest to the update scenario is shown here. As this case study focuses on Alice's Lacrosse club membership and related authorisations let us consider the following subset of Alice's full SocACL PB.

```
alice says alice.allow.?A.view."cats.jpg".social if ?A.memberOf."UoL
  Lacrosse", not ?A.memberOf."UoL Tennis", ?A != alice;
alice says alice.allow.?A.view."dogs.jpg".social if ?A.memberOf."UoL
  Lacrosse", not ?A.memberOf."UoL Coffee Lovers", ?A != alice;
alice says alice.memberOf."UoL Lacrosse";
```

Using jSocACL from Section 3.1 to translate the above produces the following ASP PB $\Pi_A =$

```
allow( alice, A, view, "cats.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Tennis" ), !=( A,
  alice ).
allow( alice, A, view, "dogs.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Coffee Lovers" ), !=(
   A, alice ).
memberOf( alice, alice, "UoL Lacrosse" ).
```

Note the above is a subset of the full translation, found in Appendix A.2.2, and includes modifications performed by jSocACL's post-process, Section 3.1.2. Based on Alice's observations in O_A she makes the assumption Bob is a member of the Lacrosse club, such that $H_A^+ =$

```
memberOf( alice, bob, "UoL Lacrosse" )
```

The UP of P_A and $\mathcal{UR}_A$ can be found in Appendix C.6.1. Interestingly, when jUpABAC attempts to solve the program through its DLVWrapper calls the API reports the DLV error:

```
DLV [build BEN+ODBC/Dec 17 2012 gcc 4.2.1 (Apple Inc. build 5666) (dot
  3)]
Aggregate function cannot be applied on disjunctive or unstratified
  predicates, if it is used in an assignment with an unbounded variable.
Error occurs in
friendCount(alice,alice,_0) :- pred15(_2), _0 <= #count{<bob:pred1(
  close_friend,bob)>} <= _0, r6.
```

This error refers to a rule which is not part of the translation, Appendix A.2.2, but rather is a product of an optimisation technique used by DLV where the user program is rewritten. Using DLV's "–print-magic" option displays this rewritten program in full. Part of this output can be found below. Full output is available in Appendix C.6.2.

```
friendCount(alice,alice,X0) :- pred17(X2), X0 <= #count{X1 : pred1(X2,
  X1)} <= X0, r6.
relationship(alice,alice,carl,coworker) :- !=(alice,carl), r14.
relationship(alice,alice,bob,close_friend) :- !=(alice,bob), r15.
r14 :- not neg_r14. neg_r14 :- not r14.
r15 :- not neg_r15. neg_r15 :- not r15.
```

DLV's rule rewriting replaces the `relationship` predicates forming the `friendCount` aggregate with `pred1` and `pred17`. Closer inspection of `pred1` and `pred17`, shown below, starts to reveal the cause of the error.

```
pred1(X2,X1) :- relationship(alice,alice,X1,X2).
pred17(X2) :- relationship(alice,alice,X3,X2).
pred7(X0,X0,X2,close_friend) :- !=(X0,X2), relationship(X0,X0,X2,
  close_friend).
pred8(X2,X2,X3,coworker) :- !=(X2,X3), relationship(X2,X2,X3,coworker).
pred9(X3,X3,X1,wife) :- !=(X3,X1), relationship(X3,X3,X1,wife).
pred10(X2,X2,X1,class_mate) :- !=(X2,X1), relationship(X2,X2,X1,
  class_mate).
```

This rewriting is done by DLV's "Magic Set Rewriter" (MSR) module which implements Dynamic Magic Sets (DMS) as introduced by Alvinao et al. [5].

DMSs are an extension of *magic sets*, a popular technique for optimising ASP solvers [10]. Using this technique ASP programs are written in such a way when viewing the solving of the program as a search tree “unnecessary” branches are ignored. Normally this technique has no impact on the user program except improving the solve time. In our usage there appears to be some unexpected interaction between UPs and the DMS technique. Consider the following rules from the rewritten UP.

```
relationship(alice,alice,bob,close_friend) :- !=(alice,bob), r15.
pred1(X2,X1) :- relationship(alice,alice,X1,X2).
r15 :- not neg_r15. neg_r15 :- not r15.
```

The indirection introduced by DMS allows the `relationship` predicate to reduce to a single disjunctive rule:

```
relationship(alice,alice,bob,close_friend) v neg_r15.
```

A similar reduction would also exist for Alice’s relationship with Carl and seems to be the cause of the error. As DLV is heavily geared towards disjunctive logic programs it seems reasonable that its optimisation techniques aim to reduce user programs to this class of programs. Problematically DLV’s implementation of aggregates does not support disjunctive predicates, hence the error.

To confirm this we attempted to run DLV with as many optimisations as possible disabled. Since DLV requires the “Body Reordering” optimisation to be enabled when built-ins, such as aggregates, are used it was left enabled. However, even with most of the optimisation tools disabled DLV still outputs a similar error:

```
DLV [build BEN+ODBC/Dec 17 2012 gcc 4.2.1 (Apple Inc. build 5666) (dot
  3)]
Aggregate function cannot be applied on disjunctive or unstratified
  predicates, if it is used in an assignment with an unbounded variable.
Error occurs in
friendCount(alice,alice,_0) :- _0 <= #count{<bob:aux#_0_1~23$1$1_1_0|2(
  close_friend,bob)>} <= _0, relationship(alice,alice,_3,_2), r6.
```

Though the above error message indicates DLV has performed less rewriting some still occurs. This result suggests that the rewriting is unavoidable and there is an inherent problem with the interaction between DLV’s aggregates and our UPs. To confirm this we re-enable DLV’s default settings and consider the other aggregate in Alice’s PB; `mostPopular`. Again using the “–print-magic” displays the following:

```
mostPopular(alice,alice,photo,X0) :- pred5(X0), pred3(X0,X3), pred6(X6)
  , pred4(X6), X3 <= #max{X4 : pred0(X6,X4)} <= X3, r8.
```

```
description(alice,X0,animalPhoto) :- pred15(X0), pred5(X0), r19.
photoOf(alice,X0,animals) :- pred4(X0), r3.
isIn(alice,"cats.jpg",animal) :- r11.
type(alice,"cats.jpg",photo) :- r22.
isIn(alice,"dogs.jpg",animal) :- r0.
type(alice,"dogs.jpg",photo) :- r12.
```

As with `friendCount`, DLV has replaced `mostPopular`'s body predicates with `pred0`, `pred3`, `pred4`, `pred6`, and `pred6`. If there is an inherent issue with our framework and DLV aggregates then one would expect `mostPopular` to produce the same errors as `friendCount` when DLV is run. To test this the `friendCount` rule is flagged in Π_A so it is ignored by jUpABAC when generating the UP. Interestingly, when DLV attempts to solve the new UP it does not report any errors, suggesting the problem is not inherent to our framework. To better understand this result let us consider the "–print-magic" output of the new UP, shown below.

```
pred0(X6,X4) :- pred3(X6,X4), pred4(X6).
pred2(X1,X0,"UoL Lacrosse") :- !=(X0,alice), memberOf(X1,X0,"UoL
  Lacrosse").
pred3(X6,X4) :- likes(X5,X6,X4).
pred4(X6) :- description(X7,X6,animalPhoto).
pred5(X0) :- type(X1,X0,photo).
pred6(X6) :- likes(X8,X6,X9).
pred7(X0,X0,X2,close_friend) :- !=(X0,X2), relationship(X0,X0,X2,
  close_friend).
pred8(X2,X2,X3,coworker) :- !=(X2,X3), relationship(X2,X2,X3,coworker).
pred9(X3,X3,X1,wife) :- !=(X3,X1), relationship(X3,X3,X1,wife).
pred10(X2,X2,X1,class_mate) :- !=(X2,X1), relationship(X2,X2,X1,
  class_mate).
pred13(X1) :- !=(X1,alice), enrolled(X0,X1,"UoL","Computer Science").
pred15(X0) :- isIn(X1,X0,animal).
pred16(X0,X1) :- memberOf(X2,X0,X1).
pred18(X3,X0) :- <=(X3,2), rindrelationship(X2,alice,X3,X0).
```

Using the above rules we attempt to perform the same reduction we did for `friendCount` to form a disjunctive rule. However, this reduction is not possible with the `mostPopular`'s related rules. With this in mind we come to the follow conclusions on DLV's aggregates and our framework's UPs. Firstly, DLV appears to perform some amount rewriting to aggregates no matter which option are set to disable them, leading to reductions which cause disjunctive predicates to occur. Secondly, the definition of UPs leads to these disjunctive predicates in certain instances. These two factor culminate in DLV's optimisation techniques and our

UP interacting in an unexpected way. This leads us to wonder how our update behaves when different ASP solvers are used. As such, we consider the interaction between our update formalism and various ASP solvers as future work.

We continue the experiments with `friendCount` flagged to be ignored to produce the following UC.

```
memberOf( alice, alice, "UoL Lacrosse" ).
allow( alice, A, view, "cats.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Tennis" ), !=( A,
  alice ).
allow( alice, A, view, "dogs.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Coffee Lovers" ), !=(
   A, alice ).
```

This UC nominates Alice's Lacrosse membership attribute, and her `cats.jpg` and `dogs.jpg` authorisation rules for removal. As this is the only UC it becomes Δ by default. Applying Δ to the original PB yields an updated PB which does not contain the above three rules. Since this result uses a lot of page space we omit the result from this section, but provide it in full in Appendix C.6.3.

Despite Alice's PB and update scenario being significantly more complex than the TV Off case study it can be seen in Table 7.2 the execution time is still acceptable. However, as with the TV Off experiments, the size of the PB is so small these results are of little interest.

7.3.3 Experiment: PolicyGen PBs

The performance of jUpABAC is evaluated using PBs generated usingour PolicyGen utility introduced in Section 3.4.1. All of these experiments consider a contraction update where a random authorisation rule is selected for removal. Assumptions are generated from this rule's Body.

For all of the PolicyGen experiments we only consider contraction updates as the removal of rules is the most computationally intensive component of our update. This is because adding new rules can be achieved through a simple set union. On the other hand, computing Δ requires the solving of potentially large ASP programs. Table 7.3 summarises these experiments and their results.

The experiments presented in Table 7.3 consider a range of different PB sizes. Small PB are considered by Gen01 to Gen04. Gen05 to Gen08 update normal sized PBs. Gen09 at around 1000 rules is what we believe is the upper limit for a ABAC PB intended for use in OSNs. Gen10 to Gen14 consider exceptionally large PBs to highlight optimisation strategies.

All experiments up to Gen07 are completed in under 3 seconds. Gen08 is

Experiment	Π	**UR** O	**UR** U	**UP**	**UC**	**CPU Time (s)**
Gen01	15	1	0	61	2	0.13
Gen02	37	1	0	149	3	0.17
Gen03	89	1	0	357	1	0.26
Gen04	74	1	0	297	1	0.22
Gen05	137	1	0	549	1	0.37
Gen06	177	1	0	709	1	0.59
Gen07	401	1	0	1605	2	2.72
Gen08	625	1	0	2501	2	7.47
Gen09	977	1	0	3909	1	22.57
Gen10	1263	1	0	5053	1	45.80
Gen11	1935	1	0	7741	2	198.20
Gen12	3748	1	0	14993	1	1660.50
Gen13	8104	1	0	32417	1	41678.31
Gen14	10868	1	0	43473	N/A	N/A

Table 7.3: PolicyGen Experiment Performance Results

noticeably slow, but at 8 seconds it can still be considered acceptable. Taking nearly 30 seconds Gen09 is slow, but depending on how frequently the PB is updated it may still be tolerable. Experiments Gen10 and onward are intolerably slow with Gen13 taking over 11 hours. For this reason Gen14 to Gen18 have not been attempted.

These results show that jUpABAC can support PBs of up to 1000 rules. Beyond this size, though slow the experiments which have 2 or more UCs suggest there is still a benefit to use jUpABAC. In the case of Gen11 with 1935 rules it is unlikely a user could manually find two removal options to prevent the unwanted outcome. For each update only one ASP program is solved which suggests the poor results of Gen10 onwards is a result of computing the UP. As previously discussed in Section 5.2.3, the performance of the solver used by jUpABAC, DLV, is outdone by other solvers. This coupled with our issues with DLV in Section 7.3.2 suggests it is beneficial for our update to use a different solver. As such, we note the investigation of alternative solvers, specifically WASP, as future work.

7.4 Chapter Summary

In this chapter we presented and evaluated jUpABAC, an implementation of the ABAC update presented in Chapter 6. In Section 7.1 we outlined the jUpABAC system. This is followed up in Section 7.2 with a detailed overview of the Java classes which form jUpABAC. The chapter concluded with Section 7.3 with a series of experiments evaluating the performance of jUpABAC.

Chapter 8

Conclusions

This chapter summarises the research presented throughout this thesis and outlines ideas for the future work.

8.1 Summary of Research

This thesis has introduced, developed, and analysed the foundations and implementations of declarative access control for OSNs. Specifically, the research has approached the important issue of OSN privacy management as an access control problem. Through the problem domain, this thesis has investigated and addressed research gaps relating to the ABAC policy specification, evaluation, and update.

As noted in Section 1.2, ABAC policy specification schemes often struggle with a trade-off between features and concise semantics [28]. There are also concerns over the evaluation of ABAC policies given the potentially conflicting nature of attributes [74]. One of the claimed benefits of ABAC over existing models is the ease of policy maintenance [51]. However, despite this there exists, to the best of our knowledge, no research into the formal update of ABAC policies. To tackle these issues the research presented throughout this thesis has developed and analysed a suite of access control formalisms based on the novel application of ASP and other logic programming techniques.

Chapter 2 begins by exploring ABAC policy expression through the development of a policy specification language called SocACL. SocACL provides a variety of statement forms which uses information readily available in OSNs as decision criteria, such as direct and indirect relationships. To address the trade-off between features and semantics noted by Crampton et al. [28] SocACL's features are underpinned by precise semantics defined as a translation between the language and ASP. A prototype implementation of this translation, called jSocACL, is presented in Chapter 3. This translation creates an equivalence between SocACL and ASP allowing for the use of logic programming techniques for the evaluation and update of ABAC policies.

In Chapter 4 this equivalence is leveraged to develop a novel approach to ABAC policy evaluation. This approach adapts the buyer/seller negotiation formalism presented by Son and Sakama [77]. Here user attributes are treated as currency by a agent acting on behalf of a user. Agents for a resource requester try to minimise the "cost" of the resource by reducing the number attributes they reveal to the resource holder. On the other hand, agents acting on behalf of the holder aims to maximise trust in the requester by asking for as many attributes as the requester will bear, effectively trying to raise the "cost" of the resource. Through a formal analysis of the formalism it was found our policy evaluation approach can support the initialisation of PBs and the common update operations of expansion, contraction, and revision. Chapter 5 introduces jNQS, a prototype implementation of our negotiation based policy evaluation formalism.

The challenge of ABAC policy update is addressed in Chapter 6 with the introduction of a ABAC policy update methodology based on logic programming update techniques. Here updates are defined as a set of observed unwanted access control outcomes and a set of new rules. Using abductive reasoning, rules causing the unwanted outcomes are found and removed. Crucially, the update finds a solution to the unwanted outcome without the need for the user to determine the cause. This has been done to address studies [65] which found OSN users struggle to correct errors with their privacy setting after being informed of them. Finally, we dedicated Chapter 7 to the introduction of an implementation of this update called jUpABAC. Here experiment results are presented about the interaction between our update's UP formalism and optimisation techniques employed by DLV.

8.2 Considerations for the Future Work

8.2.1 Obligations

Our first consideration for future work is the reintroduction of obligations to SocACL. Since obligations are a core to the OECD's guidelines on transferring personal information it is desirable that SocACL supports them. Obligations are also interesting from a wider academic standpoint. As shown in our previous work [23] the formal representation of obligations is challenging for a number of reasons. Firstly, there is a need for a means to bind an obligation to specific authorisations. Though we attempt to do this in [23], the approach is crude and syntactically counterintuitive. Secondly, the obligations in [23] do not consider the enforcement of obligations. Finally, there are various technical challenges where the resolution of some obligations has to be performed in tandem with its corresponding authorisation.

One line of investigation for this would be to consider the obligations presented by Pieter et al. [72]. They consider obligations as a component of a larger organisational security policy. By formalising their approach in FOL they are able to utilise off-the-shelf tools to perform analysis of there policies. Interestingly, Pieter et al. [72] follow a different interpretation of obligations than the OECD guidelines. Instead of considering an obligation as some pre-agreement to do something later in order to access some resource they consider obligations a refinement of a higher-level policies denoting a task agents within a system need to, or are obliged, to carry out to ensure certain security outcomes. For instance, "stealing a laptop left in an office" is prevented by having employees be obligated to lock the door of the office when leaving the laptop unattended.

8.2.2 Distributed Negotiations

Another consideration for the future work is distributed negotiation. During a query negotiation intuitively an agent should only trust some attributes asserted by an authority, such as only trusting club membership attributes when provided by a club committee member. In this case it would be desirable for an agent to be able to request attributes from other entities instead of the opponent. Currently our policy evaluation system does not support such a scheme, despite SocACL's "says" statement syntax suggesting it might. It would also be interesting to consider the ability for distributed negotiations to support some form of delegation. For instance, an agent could make assertions on behalf of other agents, or attempt to "backup" their own attributes using some sort of special attribute based on cryptographic signatures.

8.2.3 Automated Assumption Derivation

Our final consideration for future work is automated assumption derivation. As it stands, the biggest technical issue with our set of ABAC tools is the need for the user to manually determine and input the assumptions. Automated assumption generation was prototyped in PolicyGen, but, it highlighted more issues. For instance, when considering inequalities, such as `?A < 100`, the assumption set would need to consider all potential values of the variable `?A`.

In addition to this, automated assumption derivation has wider applications. Despite assumptions being a key component of the work by Sakama et al. [73] and Son et al. [77] they do not formally, or informally, outline an approach to generating assumptions. Based on our update formalism in Chapter 6 we believe it is reasonably straightforward to derive assumptions based on the set of unwanted access control outcomes. For example, given the following unwanted outcome:

```
allow(dan, carl , "write", "UoL Sports Gallery", social)
```

When the above is considered in the context of the PB which yielded it, shown below, it is easy to see that the assumptions should be. By following the implication chain created by the below rules, clearly the assumptions should consist of `carl` being a member of all of the listed clubs. Using this idea of backward chaining or using ASP grounding [11] we believe it is possible to reliably derive assumptions for the update framework.

```
allow(dan, A, "write", "UoL Sports Gallery", social) :- memberof(A, A,
  "UoL Sports").
memberof(A, A, "UoL Sports") :- memberof(A, A, "UoL Lacrosse").
memberof(A, A, "UoL Sports") :- memberof(A, A, "UoL Tennis").
memberof(A, A, "UoL Sports") :- memberof(A, A, "UoL Hockey").
memberof(A, A, "UoL Sports") :- memberof(A, A, "UoL Hotdog Eating").
```

Bibliography

[1] Variations in Access Control Logic. In R. van der Meyden and L. van der Torre, editors, *Deontic Logic in Computer Science*, volume 5076 of *Lecture Notes in Computer Science*, pages 96–109. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-70525-3_9.

[2] M. Abadi. Logic in Access Control. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 228 – 233, June 2003.

[3] C.E. Alchourrón, P. Gärdenfors, and D. Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *The Journal of Symbolic Logic*, 50(02):510–530, 1985.

[4] J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, and T.C. Przymusinski. Dynamic Logic Programming. *KR*, 98:98–109, 1998.

[5] M. Alviano and W. Faber. Dynamic Magic Sets and Super-Coherent Answer Set Programs. *AI Communications*, 24(2):125–145, 2011.

[6] C. Au Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-lee. Decentralization: The Future of Online Social Networking. In *In W3C Workshop on the Future of Social Networking Position Papers*, 2009.

[7] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An Online Social Network with User-Defined Privacy. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 135–146, New York, NY, USA, 2009. ACM.

[8] Y. Bai. On Distributed System Security. In *Security Technology, 2008. SECTECH '08. International Conference on*, pages 54–57, December 2008.

[9] Y. Bai, E. Caprin, and Y. Zhang. Reasoning about the State Change of Authorization Policies. In *Proceedings of the 28th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, Seoul, Korea, June 10-12*, 2015.

[10] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15. ACM, 1985.

[11] C. Baral. *Knowledge Representation, Reasoning and Declartive Problem Solving.* Cambridge University Press, 1st edition, 2010.

[12] C. Baral, G. Gelfond, T.C. Son, and E. Pontelli. Using Answer Set Programming to Model Multi-Agent Scenarios Involving Agents' Knowledge About Other's Knowledge. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, volume 1 of *AAMAS '10*, pages 259–266, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[13] E. Barnet. Facebook dismisses rumors of charging plans. "http://www.telegraph.co.uk/technology/facebook/6973757/Facebook-dismisses-rumours-of-charging-plans.html", January 2010.

[14] A. Baumgrass, M. Strembeck, and S. Rinderle-Ma. Deriving Role Engineering Artifacts From Business Processes and Scenario Models. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 11–20, New York, NY, USA, 2011. ACM.

[15] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A Logical Framework for Reasoning About Access Control Models. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, SACMAT '01, pages 41–52, New York, NY, USA, 2001. ACM.

[16] A. Bielenberg, L. Helm, A. Gentilucci, D. Stefanescu, and H. Zhang. The Growth of Diaspora-A Decentralized Online Social Network in the Wild. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012, IEEE Conference on*, pages 13–18. IEEE, 2012.

[17] M. Blaze, J. Ioannidis, and A. Keromytis. Experience with the Keynote Trust Management System: Applications and Future Directions. In P. Nixon and S. Terzis, editors, *Trust Management*, volume 2692 of *Lecture Notes in Computer Science*, pages 1071–1071. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-44875-6_21.

[18] D.M. Boyd and N.B. Ellison. Social Network Sites: Definition, History, and Scholarship. *Engineering Management Review, IEEE*, 38(3):16–31, 2010.

[19] G. Brewka, T. Eiter, and M. Truszczynski. Answer Set Programming at a Glance. *Commun. ACM*, 54(12):92–103, December 2011.

[20] F. Buccafurri, N. Leone, and P. Rullo. Enhancing Disjunctive Datalog by Constraints. *Knowledge and Data Engineering, IEEE Transactions on*, 12(5):845–860, 2000.

[21] E. Caprin and Y. Zhang. SocACL: An ASP-Based Access Control Language for Online Social Networks. In *Communications and Multimedia Security - 14th IFIP TC 6/TC 11 International Conference, CMS 2013, Magdeburg, Germany, September 25-26, 2013. Proceedings*, pages 207–210, 2013.

[22] E. Caprin and Y. Zhang. Negotiation Based Framework for Attribute-Based Access Control Policy Evaluation. In *Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014*, page 122, 2014.

[23] E. Caprin, Y. Zhang, and K.M. Khan. Social Access Control Language (SocACL). In *The 6th International Conference on Security of Information and Networks, SIN '13, Aksaray, Turkey, November 26-28, 2013*, pages 261–265, 2013.

[24] G.P. Cheek and M. Shehab. Policy-by-Example for Online Social Networks. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 23–32, New York, NY, USA, 2012. ACM.

[25] Y. Cheng, J. Park, and R. Sandhu. A User-to-User Relationship-Based Access Control Model for Online Social Networks. In *Data and Applications Security and Privacy XXVI*, volume 7371 of *Lecture Notes in Computer Science*, pages 8–24. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-31540-4_2.

[26] S. Chin and S.B. Older. *Access Control, Security and Trust: A Logical Approach.* Taylor and Francis Inc, 2010.

[27] E. Cohen, R.K. Thomas, W. Winsborough, and D. Shands. Models for coalition-based access control (cbac). In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 97–106, New York, NY, USA, 2002. ACM.

[28] J. Crampton and C. Morisset. PTaCL: A Language for Attribute-Based Access Control in Open Systems. *Lecture Notes in Computer Science*, 7215 LNCS:390–409, 2012.

[29] L.A. Cutillo, R. Molva, and T. Strufe. Safebook: A Privacy-Preserving Online Social Network Leveraging on Real-Life trust. *Communications Magazine, IEEE*, 47(12):94 – 101, December. 2009.

[30] I.B. Dhia. Access Control in Social Networks: A Reachability-Based Approach. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 227–232, New York, NY, USA, 2012. ACM.

[31] C. Dodaro, M. Alviano, W. Faber, N. Leone, F. Ricca, and M. Sirianni. The Birth of a WASP: Preliminary Report on a New ASP Solver. In *CILC*, pages 99–113. Citeseer, 2011.

[32] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On Updates of Logic Programs: Semantics and Properties. Technical report, Citeseer, 2000.

[33] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa. Design and Implementation of Aggregate Functions in the DLV System. *Theory and Practice of Logic Programming*, 8:545–580, 10 2008.

[34] D.F. Ferraiolo and D.R. Kuhn. Role-Based Access Control. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[35] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli. *Role-Based Access Control.* Artech House, 2nd revised edition, 2007.

[36] P.W.L. Fong. Preventing Sybil Attacks by Privilege Attenuation: A Design Principle for Social Network Systems. In *Security and Privacy, 2011, IEEE Symposium on*, pages 263 –278, May. 2011.

[37] P.W.L. Fong and I. Siahaan. Relationship-Based Access Control Policies and Their Policy Languages. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 51–60, New York, NY, USA, 2011. ACM.

[38] N. Foo and Y. Zhang. Updating Logic Programs. In *Proceedings 13th European Conference on Artificial Intelligence (ECAI 1998)*, pages 403–407. Citeseer, 1998.

[39] H. Gao, J. Hu, T. Huang, J. Wang, and Y. Chen. Security Issues in Online Social Networks. *Internet Computing, IEEE*, 15(4):56–63, July-August 2011.

[40] D. Garg and F. Pfenning. Non-Interference in Constructive Authorization Logic. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, page 11 pp., 2006.

[41] M. Gebser, R. Kaminski, B. Faufmann, M. Ostrowski, T. Schab, and S. Thiele. *A User's Guide to gringo, clasp, clingo and iclingo.* University of Potsdam, 2008.

[42] M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. In *5th Int. Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[43] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3-4):365–385, 1991.

[44] V. Genovese and D. Garg. New Modalities for Access Control Logics: Permission, Control and Ratification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7170 LNCS:56–71, 2012.

[45] H.J. Gensler. *Introduction to Logic.* Taylor and Francis Inc, 2nd revised edition, 2010.

[46] F. Giunchiglia, R. Zhang, and B. Crispo. RelBAC: Relation Based Access Control. In *Semantics, Knowledge and Grid, 2008. SKG '08. 4th International Conference on*, pages 3 –11, December. 2008.

[47] G. Goncalves and A. Poniszewska-Maranda. Role Engineering: From Design to Evolution of Security Schemes. *Journal of Systems and Software*, 81(8):1306 – 1326, 2008.

[48] R. Gross and A. Acquisti. Information Revelation and Privacy in Online Social Networks. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 71–80. ACM, 2005.

[49] Y. Gurevich and I. Neeman. Logic of Infons: The Propositional Case. *ACM Trans. Comput. Logic*, 12(2):9:1–9:28, January 2011.

[50] H. Hu, G. Ahn, and J. Jorgensen. Multiparty Access Control for Online Social Networks: Model and Mechanisms. *Knowledge and Data Engineering, IEEE Trans. on*, 25(7):1614–1627, 2013.

[51] V.C. Hu, D.F. Ferraiolo, D.R. Kuhn, A.R. Friedman, A.J. Lang, M.M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, and Others. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, 800:162, 2014.

[52] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, 2nd revised edition, 2004.

[53] DIASPORA Inc. Diaspora. http://joindiaspora.com, 2012.

[54] S. Jahid, P. Mittal, and N. Borisov. EASiER: Encryption-Based Access Control in Social Networks with Efficient Revocation. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 411–415, New York, NY, USA, 2011. ACM.

[55] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Trans. Database Syst.*, 26(2):214–260, June 2001.

[56] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards Formal Verification of Role-Based Access Control Policies. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):242–255, Oct.-Dec. 2008.

[57] P. Jin and Y. Fang-chun. Description Logic Modeling of Temporal Attribute-Based Access Control. In *Communications and Electronics, 2006. ICCE '06. First International Conference on*, pages 414–418, oct. 2006.

[58] L. Kagal and H. Abelson. Access Control is an Inadequate Framework for Privacy Protection. *W3C Privacy Workshop*, (June):1–6, 2010.

[59] D.R. Kuhn, E.J. Coyne, and T.R. Weil. Adding Attributes to Role-Based Access Control. *Computer*, 43(6):79 –81, June 2010.

[60] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006.

[61] J. Li, N. Li, and W. Winsborough. Automated Trust Negotiation Using Cryptographic Credentials. *ACM Trans. Inf. Syst. Secur.*, 13(1):2:1–2:35, November 2009.

[62] N. Li, B.N. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-Based Approach to Distributed Authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1):128–171, February 2003.

[63] N. Li, J.C. Mitchell, and W. Winsborough. Design of a role-Based Trust-Management Framework. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 114–130, 2002.

[64] H.R. Lipford, A. Besmer, and J. Watson. Understanding Privacy Settings in Facebook with an Audience View. In *Proc. of the 1st Conf. on Usability, Psychology, and Sec.*, UPSEC'08, pages 2:1–2:8, Berkeley, CA, USA, 2008. USENIX Association.

[65] M. Madejski, M. Johnson, and S.M. Bellovin. A Study of Privacy Settings Errors in an Online Social Network. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pages 340–345, march 2012.

[66] A.K. Malik and S. Dustdar. Sharing and Privacy-Aware RBAC in Online Social Networks. In *Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE 3rd International Conference on and 2011 IEEE 3rd International Conference on Social Computing (SocialCom)*, pages 1352–1355, October. 2011.

[67] E.E. Mon and T.T. Naing. The Privacy-Aware Access Control System Using Attribute-and Role-Based Access Control in Private Cloud. In *Broadband Network and Multimedia Technology (IC-BNMT), 2011 4th IEEE International Conference on*, pages 447–451, October. 2011.

[68] 3 News. Australian PM caught following porn star. "http://www.3news.co.nz/Australian-PM-caught-following-porn-star/tabid/209/articleID/135848/Default.aspx", 2010.

[69] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C. Karat, J. Karat, and A. Trombeta. Privacy-Aware Role-Based Access Control. *ACM Trans. Inf. Syst. Secur.st. Secur.*, 13(3):24:1–24:31, July 2010.

[70] OECD. OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data. "http://www.oecd.org/", September 1980.

[71] T. Parr. ANTLR3 project web page. http://www.antlr3.org, 2014.

[72] W. Pieters, J. Padget, F. Dechesne, V. Dignum, and H. Aldewereld. Obligations to Enforce Prohibitions: On the Adequacy of Security Policies. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN '13, pages 54–61, New York, NY, USA, 2013. ACM.

[73] K. Sakama, C.and Inoue. An Abductive Framework for Computing Knowledge Base Updates. *Theory and Practice of Logic Programming*, 3(06):671–715, 2003.

[74] R. Sandhu. The Authorization Leap from Rights to Attributes: Maturation or Chaos? In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 69–70, New York, NY, USA, 2012. ACM.

[75] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.

[76] C. Schulze, L. Schöler, and B. Skiera. Not All Fun and Games: Viral Marketing for Utilitarian Products. *Journal of Marketing*, 78(1):1–19, 2014.

[77] T.C. Son and C. Sakama. Negotiation Using Logic Programming with Consistency Restoring Rules. In *IJCAI*, pages 930–935, 2009.

[78] A. Squicciarini and S. Sundareswaran. Web-Traveler Policies for Images on Social Networks. *World Wide Web*, 12:461–484, 2009. 10.1007/s11280-009-0070-8.

[79] DLVSYSTEM s.r.l. DLV System. http://www.dlvsystem.com, 2012.

[80] Sydney Morning Herald. Fake Facebook page: ex-lover to face trial for indentify theft. "http://www.smh.com.au/technology/technology-news/fake-facebook-page-exlover-to-face-trial-for-identity-theft-20111104-1myqq.html", November 2011.

[81] Sydney Morning Herald. Unreal: Facebook reveals 83 million fake profiles. "http://www.smh.com.au/world/unreal-facebook-reveals-83-million-fake-profiles-20120803-23kzj.html", Auguest 2012.

[82] The Australian. Gun-toting swim stars Nick D'Arcy and Kenrick Monk apologise. "http://www.theaustralian.com.au/sport/gun-toting-swim-stars-nick-darcy-and-kenrick-monk-apologise/story-e6frg7mf-1226388646969", June 2012.

[83] J.R. Vacca, editor. *Computer and Information Security Handbook*. Elsevier Science and Technology, 2009.

[84] L. Wang, D. Wijesekera, and S. Jajodia. A Logic-Based Framework for Attribute Based Access Control. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, FMSE '04, pages 45–55, New York, NY, USA, 2004. ACM.

[85] S. Wang and Y. Zhang. Handling Distributed Authorization with Delegation Through Answer Set Programming. *International Journal of Information Security*, 6:27–46, 2007. 10.1007/s10207-006-0008-4.

[86] E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. In *Web Services, 2005. ICWS 2005.Proceedings. 2005 IEEE International Conference on*, pages 2 vol. (xxxiii+856), July 2005.

Appendix

Appendix A

Chapter 2 Examples

A.1 Universal Additions (UA)

```
action(P, Prin, Act, Obj, Pu) :- allow(Prin, P, Act, Obj, Pu), not deny
  (Prin, P, Act, Obj, Pu).
path(X,Y,1) :- relationship(X,X,Y,R), not #int(R).
path(X,Z,D1) :- path(X,Y,D), path(Y,Z,1), +(D,1,D1), !=( X, Y ), !=( Y,
   Z ), !=( X, Z ).
rindRelationship(X,X,Y,D) :- D = #min { D1 : path(X,Y,D1) }, path(X,Y,D
  ).
```

A.2 Alice

A.2.1 SocACL PB

```
alice says alice.married;
alice says alice.hair_colour.brown;
alice says ?A.isIn.public if ?A.isIn.animal;
alice says ?A.isIn.public if ?A.isIn.plant;
alice says ?A.isIn.gallery if ?A.isIn.public;
alice says ?A.isIn.gallery if ?A.isIn.private;
alice says alice.relationship.close_friend.bob;
alice says alice.relationship.coworker.carl;
alice says define.relchain.ccm.(coworker, class_mate);
alice says define.relchain.ccw.(close_friend, coworker, wife);
alice says alice.friendCount.?A if ?A = count.(?Sub).(alice says alice.
  relationship.?Any.?Sub);
alice says alice.mostPopular.photo.?Object if ?Object.type.photo, ?
  Object.likes.?V, ?V=max.(?L).(?Obj.likes.?L, ?Obj.description.
  animalPhoto);
alice says define.description.plantPhoto.?Object.(?Object.isIn.plant,?
  Object.type.photo);
alice says ?Object.photoOf.animals if ?Object.description.animalPhoto;
alice says allow.?Other.view.?Object.social if alice.rindRelationship.?
  A.Other, ?A <= 2, Object.description.plantPhoto;
alice says alice.memberOf."UoL Lacrosse";
alice says "cats.jpg".isIn.animal;
alice says "dogs.jpg".isIn.animal;
alice says "cats.jpg".type.photo;
alice says "dogs.jpg".type.photo;
alice says "cactus.jpg".isIn.plant;
alice says "cactus.jpg".type.photo;
alice says "holiday.mov".isIn.private;
alice says "holiday.mov".type.video;
```

A.2.2 ASP PB

```
married( alice, alice ).
hair_colour( alice, alice, brown ).
isIn( alice, A, public ) :- isIn( _, A, animal ).
isIn( alice, A, public ) :- isIn( _, A, plant ).
isIn( alice, A, gallery ) :- isIn( _, A, public ).
isIn( alice, A, gallery ) :- isIn( _, A, private ).
relationship( alice, alice, bob, close_friend ) :- !=( alice, bob ).
relationship( alice, alice, carl, coworker ) :- !=( alice, carl ).
friendCount( alice, alice, A ) :- A = #count { Sub : relationship(
  alice, alice, Sub, Any ) }, relationship( alice, alice, Sub1, Any ).
mostPopular( alice, alice, photo, Object ) :- type( _, Object, photo ),
   likes( _, Object, V ), V = #max { L : likes( _, Obj, L ), description
  ( _, Obj, animalPhoto ) }, likes( _, Obj, L1 ), description( _, Obj,
  animalPhoto ).
photoOf( alice, Object, animals ) :- description( _, Object,
  animalPhoto ).
allow( alice, Other, view, Object, social ) :- rindRelationship( _,
  alice, Other, A ), <=( A, 2 ), description( _, Object, plantPhoto ).
memberOf( alice, alice, "UoL Lacrosse" ).
isIn( alice, "cats.jpg", animal ).
isIn( alice, "dogs.jpg", animal ).
type( alice, "cats.jpg", photo ).
type( alice, "dogs.jpg", photo ).
```

```
isIn( alice, "cactus.jpg", plant ).
type( alice, "cactus.jpg", photo ).
isIn( alice, "holiday.mov", private ).
type( alice, "holiday.mov", video ).
sindRelationship( alice, Sub0, Sub2, ccm ) :- relationship( Sub0, Sub0,
    Sub1, coworker ), relationship( Sub1, Sub1, Sub2, class_mate ), !=(
  Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ).
sindRelationship( alice, Sub0, Sub3, ccw ) :- relationship( Sub0, Sub0,
    Sub1, close_friend ), relationship( Sub1, Sub1, Sub2, coworker ),
  relationship( Sub2, Sub2, Sub3, wife ), !=( Sub0, Sub1 ), !=( Sub0,
  Sub2 ), !=( Sub0, Sub3 ), !=( Sub1, Sub2 ), !=( Sub1, Sub3 ), !=( Sub2
  , Sub3 ).
description( alice, Object, plantPhoto ) :- isIn( _, Object, plant ),
  type( _, Object, photo ).
```

A.3 Bob

A.3.1 SocACL PB

```
bob says bob.relationship.girlfriend.alice;
bob says bob.relationship.coworker.dan;
bob says define.relchain.cocoworker.(coworker,coworker);
bob says bob.memberOf."UoL Lacrosse" if ?A.enrolled."UoL".?Any, ?A !=
  bob;
bob says bob.memberOf."UoL Coffee Lovers" if ?A.memberOf."UoL Lacrosse
  ", ?A != alice;
```

A.3.2 ASP PB

```
relationship( bob, bob, alice, girlfriend ) :- !=( bob, alice ).
relationship( bob, bob, dan, coworker ) :- !=( bob, dan ).
memberOf( bob, bob, "UoL Lacrosse" ) :- enrolled( _, A, "UoL", Any ),
  !=( A, bob ).
memberOf( bob, bob, "UoL Coffee Lovers" ) :- memberOf( _, A, "UoL
  Lacrosse" ), !=( A, alice ).
sindRelationship( bob, Sub0, Sub2, cocoworker ) :- relationship( Sub0,
  Sub0, Sub1, coworker ), relationship( Sub1, Sub1, Sub2, coworker ),
  !=( Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ).
```

A.4 Carl

A.4.1 SocACL PB

```
carl says carl.relationship.friend.alice;
carl says carl.relationship.coworker.alice;
carl says carl.relationship.class_mate.dan;
```

A.4.2 ASP PB

```
carl says carl.relationship.friend.alice;
carl says carl.relationship.coworker.alice;
carl says carl.relationship.class_mate.dan;
```

A.5 Dan

A.5.1 SocACL PB

```
dan says dan.relationship.wife.ellen;
dan says dan.relationship.coworker.bob;
```

A.5.2 ASP PB

```
relationship( dan, dan, ellen, wife ) :- !=( dan, ellen ).
relationship( dan, dan, bob, coworker ) :- !=( dan, bob ).
```

A.6 Ellen

A.6.1 SocACL PB

```
ellen says ellen.relationship.husband.dan;
```

A.6.2 ASP PB

```
relationship( ellen, ellen, dan, husband ) :- !=( ellen, dan ).
```

Appendix B

Chapter 4 Examples

B.1 Chapter's Running Example

B.1.1 Alice's NKB $\langle \Pi_{Alice}, H^{+}_{Alice}, H^{-}_{Alice} \rangle$

Π_{Alice}

```
allow(alice, A, view, "cats.jpg", social) :- memberof(_, A, "UoL
  Lacrosse"),not memberof(_, A, "UoL Tennis"), A != alice.
allow(alice, A, view, "dogs.jpg", social) :- memberof(_, A, "UoL
  Lacrosse"),not memberof(_, A, "UoL Coffee Lovers"), A != alice.
enrolled(alice, alice, "UoL", "Computer Science") :- enrolled(_, A, "
  UoL", "Computer Science"), not memberof(_, A, "UoL Robotics"), A !=
  alice.
memberof(alice, alice, "UoL Lacrosse").
```

H^{+}_{Alice}

```
memberof(bob, bob, "UoL Lacrosse"), enrolled(bob, bob, "UoL", "Computer
   Science")
```

H^{-}_{Alice}

```
memberof(bob, bob, "UoL Coffee Lovers"), memberof(bob, bob, "UoL Tennis
  "), memberof(bob, bob, "UoL Robotics")
```

B.1.2 Bob's NKB $\langle \Pi_{Bob}, H^{+}_{Bob}, H^{-}_{Bob} \rangle$

Π_{Bob}

```
memberof(bob, bob, "UoL Lacrosse") :- enrolled(_, A, "UoL", _), A !=bob.
memberof(bob, bob, "UoL Coffee Lovers") :- memberof(_, A, "UoL Lacrosse"), A != bob.
enrolled(bob, bob, "UoL", "Computer Science").
```

H^{+}_{Bob}

```
enrolled(alice, "UoL", "Mathematics"), enrolled(alice, "UoL", "ComputerScience"),
memberof(alice, "UoL Lacrosse")
```

B.2 Son and Sakama Example

B.2.1 Seller's NKB $\langle \Pi_{Seller}, H^{+}_{Seller}, H^{-}_{Seller} \rangle$

Π_{Seller}

```
whole_sale_customer :- registered.
student_customer :- student.
senior_customer :- age(A), A >= 65.
high_pr.
low_pr :- senior_customer.
low_pr :- student_customer, good_credit.
low_pr :- student_customer, pay_cash.
lowest_pr :- whole_sale_customer, quantity(A), A >= 100.
make(A) :- product(A, _, _).
madeIn( A ) :- product(_, A, _).
colour(A) :- product(_, _, A).
product("Top Lacrosse", "France", yellow) _ product("Lacrosse Tech", "Austria", blue) _ product("Ball-o-Rama", "New Zealand", yellow) :- high_pr, not low_pr.
product("Lacrosse Tech", "Austria", blue) :- low_pr, not lowest_pr.
product("Ball-o-Rama", "New Zealand", yellow) :- lowest_pr.
```

H^+_{Seller}

```
registered, student, age(65), good_credit, pay_cash, quantity(100)
```

B.2.2 Buyer's NKB $\langle \Pi_{Buyer}, H^+_{Buyer}, H^-_{Buyer} \rangle$

Π_{Buyer}

```
age(25). student. pay_cash. quantity(1).
sale :- make("Top Lacrosse"), madeIn("France"), not
color(blue), high_pr.
sale :- make("Lacrosse Tech"), not madeIn("Australia"),low_pr.
sale :- make("Ball-o-Rama"), color(tangerine), low_pr.
sale :- make("Econocrosse"), lowest_pr.
```

H^+_{Buyer}

```
make("Top Lacrosse"), make("Lacrosse Tech"), make("Ball-o-Rama"), make
  ("Econocrosse"),
madeIn("France"), high_pr, low_pr, lowest_pr, color(tangerine)
```

H^-_{Buyer}

```
madeIn("Australia"), color(blue)
```

Appendix C

Chapter 6 Examples

C.1 Chapter Introduction Example

```
alice says alice.allow.A.view."cats.jpg".social if A.memberOf."UoL
  Lacrosse", not A.memberOf."UoL Tennis" , A != alice;
alice says alice.allow.A.view."dogs.jpg".social if A.memberOf."UoL
  Lacrosse", not A.memberOf."UoL Coffee Lovers", A != alice;
alice says alice.memberOf."UoL Lacrosse";
```

C.2 Dan's PB, Chapter Subset

$\Pi =$

```
allow(dan, A, "write", "UoL Sports Gallery") :- memberOf(A, A, "UoL
  Sports").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hotdog Eating").
```

$H^+ =$

```
memberOf(carl, carl, "UoL Lacrosse")
memberOf(carl, carl, "UoL Tennis")
memberOf(carl, carl, "UoL Hockey")
memberOf(carl, carl, "UoL Hotdog Eating")
```

C.3 UR for Dan's PB

$O =$

```
allow(carl, carl, "write", "UoL Sports Gallery")
```

$U =$

```
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming").
```

C.4 Dan's Update Program

```
:- allow(dan, carl, "write", "UoL Sports Gallery").
allow(dan, A, "write", "UoL Sports Gallery") :- memberOf(A, A, "UoL
  Sports"), r0.
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse"), r1.
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis"), r2.
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hotdog Eating"),r3.
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey"), r4.
r0 :- not neg_r0. r1 :- not neg_r1. r2 :- not neg_r2.
r3 :- not neg_r3. r4 :- not neg_r4. neg_r0 :- not r0.
neg_r1 :- not r1. neg_r2 :- not r2. neg_r3 :- not r3,
neg_r4 :- not r4.
:~ not r0. :~ not r1. :~ not r2. :~ not r3. :~ not r4.
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming" ).
```

C.5 Dan's Updated PB, Chapter Subset

```
allow(dan, A, "write", "UoL Sports Gallery") :- memberOf(A, A, "UoL
  Sports").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Lacrosse").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Tennis").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Hockey").
memberOf(A, A, "UoL Sports") :- memberOf(A, A, "UoL Swimming").
```

C.6 Experiment: Case Study Alice, Revision

C.6.1 Update Program of P_A and $\mathcal{UR}_\mathcal{A}$

```
:~ not r19. r9 :- not neg_r9.
neg_r15 :- not r15. :~ not r3. :~ not r10. neg_r14 :- not r14.
allow( alice, A, view, "cats.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Tennis" ), !=( A,
  alice ), r24.
mostPopular( alice, alice, photo, Object ) :- type( _, Object, photo ),
   likes( _, Object, V ), V = #max { L : likes( _, Obj, L ), description
  ( _, Obj, animalPhoto ) }, likes( _, Obj, L1 ), description( _, Obj,
  animalPhoto ), r8.
r16 :- not neg_r16. :~ not r21. r27 :- not neg_r27. neg_r1 :- not r1.
sindRelationship( alice, Sub0, Sub3, ccw ) :- relationship( Sub0, Sub0,
   Sub1, close_friend ), relationship( Sub1, Sub1, Sub2, coworker ),
  relationship( Sub2, Sub2, Sub3, wife ), !=( Sub0, Sub1 ), !=( Sub0,
  Sub2 ), !=( Sub0, Sub3 ), !=( Sub1, Sub2 ), !=( Sub1, Sub3 ), !=( Sub2
  , Sub3 ), r10.
sindRelationship( alice, Sub0, Sub2, ccm ) :- relationship( Sub0, Sub0,
   Sub1, coworker ), relationship( Sub1, Sub1, Sub2, class_mate ), !=(
  Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ), r27.
r13 :- not neg_r13. r26 :- not neg_r26. r4 :- not neg_r4. :~ not r11.
:~ not r2. r25 :- not neg_r25. neg_r17 :- not r17. r8 :- not neg_r8.
isIn( alice, A, gallery ) :- isIn( _, A, private ), r18.
isIn( alice, A, public ) :- isIn( _, A, plant ), r9.
photoOf( alice, Object, animals ) :- description( _, Object,
  animalPhoto ), r3.
enrolled( alice, alice, "UoL", "Computer Science" ) :- enrolled( _, A,
  "UoL", "Computer Science" ), not hh_memberOf( A, "UoL Robotics" ), !=(
   A, alice ), r26.
:~ not r20. neg_r19 :- not r19. neg_r5 :- not r5. neg_r25 :- not r25.
r15 :- not neg_r15. :~ not r1. neg_r21 :- not r21. neg_r26 :- not r26.
:~ not r12. :~ not r27. r11 :- not neg_r11. r3 :- not neg_r3.
neg_r12 :- not r12. r12 :- not neg_r12. neg_r4 :- not r4. r0 :- not
  neg_r0.
r17 :- not neg_r17. :~ not r9. neg_r6 :- not r6. r5 :- not neg_r5.
relationship( alice, alice, bob, close_friend ) :- !=( alice, bob ),
  r15.
:~ not r13. :~ not r0. r22 :- not neg_r22. :~ not r26.
isIn( alice, A, gallery ) :- isIn( _, A, public ), r2.
isIn( alice, A, public ) :- isIn( _, A, animal ), r20.
```

```
r2 :- not neg_r2. neg_r11 :- not r11.
hh_memberOf( HH0, HH1 ) :- memberOf( _, HH0, HH1 ), r23.
r21 :- not neg_r21. neg_r10 :- not r10. r7 :- not neg_r7. :~ not r8.
:~ not r25. :~ not r7. r24 :- not neg_r24. :~ not r14.
relationship( alice, alice, carl, coworker ) :- !=( alice, carl ), r14.
isIn( alice, "cats.jpg", animal ) :- r11.
type( alice, "cats.jpg", photo ) :- r22.
neg_r16 :- not r16. neg_r18 :- not r18. neg_r27 :- not r27. neg_r3 :-
  not r3.
:- memberOf( alice, alice, "UoL Lacrosse" ).
allow( alice, A, view, "dogs.jpg", social ) :- memberOf( SODFIX_0, A, "
  UoL Lacrosse" ), not memberOf( SODFIX_0, A, "UoL Coffee Lovers" ), !=(
   A, alice ), r16.
neg_r13 :- not r13. r20 :- not neg_r20. :~ not r24. :~ not r6.
:~ not r15. r18 :- not neg_r18. :~ not r17. :~ not r16.
description( alice, Object, animalPhoto ) :- isIn( _, Object, animal ),
   type( _, Object, photo ), r19.
married( alice, alice ) :- r21.
isIn( alice, "cactus.jpg", plant ) :- r1.
type( alice, "holiday.mov", video ) :- r4.
memberOf( alice, alice, "UoL Lacrosse" ) :- r5.
:- allow( alice, bob, view, "dogs.jpg", social ).
friendCount( alice, alice, A ) :- A = #count { Sub : relationship(
  alice, alice, Sub, Any ) }, relationship( alice, alice, Sub1, Any ),
  r6.
hair_colour( alice, alice, brown ) :- r7.
:~ not r23. r1 :- not neg_r1. :~ not r5. r10 :- not neg_r10.
allow( alice, Other, view, Object, social ) :- rindrelationship( _,
  alice, A, Other ), <=( A, 2 ), description( _, Object, animalPhoto ),
  r25.
neg_r2 :- not r2. neg_r7 :- not r7. neg_r20 :- not r20. r19 :- not
  neg_r19.
memberOf( alice, alice, "UoL Hockey" ).
isIn( alice, "dogs.jpg", animal ) :- r0.
isIn( alice, "holiday.mov", private ) :- r13.
neg_r23 :- not r23. r6 :- not neg_r6. neg_r8 :- not r8. neg_r0 :- not
  r0.
type( alice, "dogs.jpg", photo ) :- r12.
:~ not r18.:~ not r22. neg_r22 :- not r22. :~ not r4.
:- allow( alice, bob, view, "cats.jpg", social ).
```

```
neg_r24 :- not r24. neg_r9 :- not r9. r14 :- not neg_r14. r23 :- not
  neg_r23.
type( alice, "cactus.jpg", photo ) :- r17.
```

C.6.2 Alice's Update Program, DLV –print-magic output

```
DLV [build BEN+ODBC/Dec 17 2012 gcc 4.2.1 (Apple Inc. build 5666) (dot
  3)]

r9 :- not neg_r9. neg_r14 :- not r14. r16 :- not neg_r16. neg_r15 :-
  not r15.
allow(alice,X0,view,"cats.jpg",social) :- pred2(X1,X0,"UoL Lacrosse"),
  r24, not memberOf(X1,X0,"UoL Tennis").
pred0(X6,X4) :- pred3(X6,X4), pred4(X6).
mostPopular(alice,alice,photo,X0) :- pred5(X0), pred3(X0,X3), pred6(X6)
  , pred4(X6), X3 <= #max{X4 : pred0(X6,X4)} <= X3, r8.
sindRelationship(alice,X0,X1,ccw) :- pred7(X0,X0,X2,close_friend),
  pred8(X2,X2,X3,coworker), pred9(X3,X3,X1,wife), !=(X0,X3), !=(X0,X1),
  !=(X2,X1), r10.
sindRelationship(alice,X0,X1,ccm) :- pred8(X0,X0,X2,coworker), pred10(
  X2,X2,X1,class_mate), !=(X0,X1), r27.
r27 :- not neg_r27. neg_r1 :- not r1. r13 :- not neg_r13. r26 :- not
  neg_r26.
r4 :- not neg_r4. r25 :- not neg_r25. neg_r17 :- not r17. r8 :- not
  neg_r8.
isIn(alice,X0,gallery) :- pred11(X0), r18.
isIn(alice,X0,public) :- pred12(X0), r9.
photoOf(alice,X0,animals) :- pred4(X0), r3.
enrolled(alice,alice,"UoL","Computer Science") :- pred13(X1), r26, not
  hh_memberOf(X1,"UoL Robotics").
neg_r19 :- not r19. neg_r5 :- not r5. neg_r25 :- not r25. r15 :- not
  neg_r15.
neg_r21 :- not r21. neg_r26 :- not r26. r11 :- not neg_r11. r3 :- not
  neg_r3.
neg_r12 :- not r12. r12 :- not neg_r12. neg_r4 :- not r4. r0 :- not
  neg_r0.
r17 :- not neg_r17. neg_r6 :- not r6. r5 :- not neg_r5. r22 :- not
  neg_r22.
relationship(alice,alice,bob,close_friend) :- !=(alice,bob), r15.
isIn(alice,X0,gallery) :- pred14(X0), r2.
isIn(alice,X0,public) :- pred15(X0), r20.
```

```
r2 :- not neg_r2. neg_r11 :- not r11. r21 :- not neg_r21. neg_r10 :- not r10.
hh_memberOf(X0,X1) :- pred16(X0,X1), r23.
r7 :- not neg_r7. r24 :- not neg_r24. neg_r16 :- not r16. neg_r18 :- not r18.
relationship(alice,alice,carl,coworker) :- !=(alice,carl), r14.
isIn(alice,"cats.jpg",animal) :- r11.
type(alice,"cats.jpg",photo) :- r22.
neg_r27 :- not r27. neg_r3 :- not r3. neg_r13 :- not r13. r20 :- not neg_r20.
allow(alice,X0,view,"dogs.jpg",social) :- pred2(X1,X0,"UoL Lacrosse"), r16, not memberOf(X1,X0,"UoL Coffee Lovers").
description(alice,X0,animalPhoto) :- pred15(X0), pred5(X0), r19.
married(alice,alice) :- r21.
isIn(alice,"cactus.jpg",plant) :- r1.
type(alice,"holiday.mov",video) :- r4.
r18 :- not neg_r18. r1 :- not neg_r1. r10 :- not neg_r10. neg_r2 :- not r2.
memberOf(alice,alice,"UoL Lacrosse") :- r5.
pred1(X2,X1) :- relationship(alice,alice,X1,X2).
friendCount(alice,alice,X0) :- pred17(X2), X0 <= #count{X1 : pred1(X2, X1)} <= X0, r6.
hair_colour(alice,alice,brown) :- r7.
allow(alice,X0,view,X1,social) :- pred18(X3,X0), pred4(X1), r25.
neg_r7 :- not r7. neg_r20 :- not r20. r19 :- not neg_r19. neg_r23 :- not r23.
memberOf(alice,alice,"UoL Hockey").
isIn(alice,"dogs.jpg",animal) :- r0.
isIn(alice,"holiday.mov",private) :- r13.
r6 :- not neg_r6. neg_r8 :- not r8. neg_r0 :- not r0. neg_r22 :- not r22.
type(alice,"dogs.jpg",photo) :- r12.
neg_r24 :- not r24. neg_r9 :- not r9. r14 :- not neg_r14. r23 :- not neg_r23.
type(alice,"cactus.jpg",photo) :- r17.
pred2(X1,X0,"UoL Lacrosse") :- !=(X0,alice), memberOf(X1,X0,"UoL Lacrosse").
pred3(X6,X4) :- likes(X5,X6,X4).
pred4(X6) :- description(X7,X6,animalPhoto).
pred5(X0) :- type(X1,X0,photo).
pred6(X6) :- likes(X8,X6,X9).
```

```
pred7(X0,X0,X2,close_friend) :- !=(X0,X2), relationship(X0,X0,X2,
  close_friend).
pred8(X2,X2,X3,coworker) :- !=(X2,X3), relationship(X2,X2,X3,coworker).
pred9(X3,X3,X1,wife) :- !=(X3,X1), relationship(X3,X3,X1,wife).
pred10(X2,X2,X1,class_mate) :- !=(X2,X1), relationship(X2,X2,X1,
  class_mate).
pred11(X0) :- isIn(X1,X0,private).
pred12(X0) :- isIn(X1,X0,plant).
pred13(X1) :- !=(X1,alice), enrolled(X0,X1,"UoL","Computer Science").
pred14(X0) :- isIn(X1,X0,public).
pred15(X0) :- isIn(X1,X0,animal).
pred16(X0,X1) :- memberOf(X2,X0,X1).
pred17(X2) :- relationship(alice,alice,X3,X2).
pred18(X3,X0) :- <=(X3,2), rindrelationship(X2,alice,X3,X0).
:- memberOf(alice,alice,"UoL Lacrosse").
:- allow(alice,bob,view,"dogs.jpg",social).
:- allow(alice,bob,view,"cats.jpg",social).
:~ not r19. [1:1] :~ not r10. [1:1] :~ not r3. [1:1] :~ not r21. [1:1]
:~ not r11. [1:1] :~ not r2. [1:1] :~ not r20. [1:1] :~ not r1. [1:1]
:~ not r12. [1:1] :~ not r27. [1:1] :~ not r9. [1:1] :~ not r13. [1:1]
:~ not r0. [1:1] :~ not r26. [1:1] :~ not r8. [1:1] :~ not r25. [1:1]
:~ not r7. [1:1] :~ not r14. [1:1] :~ not r24. [1:1] :~ not r6. [1:1]
:~ not r15. [1:1] :~ not r17. [1:1] :~ not r16. [1:1] :~ not r23. [1:1]
:~ not r5. [1:1] :~ not r18. [1:1] :~ not r22. [1:1] :~ not r4. [1:1]
```

C.6.3 Alice's Updated ASP PB

```
isIn( alice, "dogs.jpg", animal ).
isIn( alice, "cactus.jpg", plant ).
isIn( alice, A, gallery ) :- isIn( _, A, public ).
photoOf( alice, Object, animals ) :- description( _, Object,
  animalPhoto ).
type( alice, "holiday.mov", video ).
hair_colour( alice, alice, brown ).
mostPopular( alice, alice, photo, Object ) :- type( _, Object, photo ),
   likes( _, Object, V ), V = #max { L : likes( _, Obj, L ), description
  ( _, Obj, animalPhoto ) }, likes( _, Obj, L1 ), description( _, Obj,
  animalPhoto ).
isIn( alice, A, public ) :- isIn( _, A, plant ).
sindRelationship( alice, Sub0, Sub3, ccw ) :- relationship( Sub0, Sub0,
   Sub1, close_friend ), relationship( Sub1, Sub1, Sub2, coworker ),
  relationship( Sub2, Sub2, Sub3, wife ), !=( Sub0, Sub1 ), !=( Sub0,
  Sub2 ), !=( Sub0, Sub3 ), !=( Sub1, Sub2 ), !=( Sub1, Sub3 ), !=( Sub2
  , Sub3 ).
isIn( alice, "cats.jpg", animal ).
type( alice, "dogs.jpg", photo ).
isIn( alice, "holiday.mov", private ).
relationship( alice, alice, carl, coworker ) :- !=( alice, carl ).
relationship( alice, alice, bob, close_friend ) :- !=( alice, bob ).
type( alice, "cactus.jpg", photo ).
isIn( alice, A, gallery ) :- isIn( _, A, private ).
description( alice, Object, animalPhoto ) :- isIn( _, Object, animal ),
   type( _, Object, photo ).
memberOf( alice, alice, "UoL Hockey" ).
isIn( alice, A, public ) :- isIn( _, A, animal ).
married( alice, alice ).
type( alice, "cats.jpg", photo ).
hh_memberOf( HH0, HH1 ) :- memberOf( _, HH0, HH1 ).
allow( alice, Other, view, Object, social ) :- rindrelationship( _,
  alice, A, Other ), <=( A, 2 ), description( _, Object, animalPhoto ).
enrolled( alice, alice, "UoL", "Computer Science" ) :- enrolled( _, A,
  "UoL", "Computer Science" ), not hh_memberOf( A, "UoL Robotics" ), !=(
   A, alice ).
sindRelationship( alice, Sub0, Sub2, ccm ) :- relationship( Sub0, Sub0,
   Sub1, coworker ), relationship( Sub1, Sub1, Sub2, class_mate ), !=(
  Sub0, Sub1 ), !=( Sub0, Sub2 ), !=( Sub1, Sub2 ).
```

Appendix D

PolicyGen Configurations and Results

D.1 Configuration File

Config.	Friend						Attributes					Descriptions		Authorisations					Policy Authority		
	Count	Types	F_Lim	T_Lim	Infer_Count	Infer_Max_Body	Count	Count_Val	Count_Val_Max	Infer_Count	Infer_Body_Max	Count	Body_Limit	Count	Body_Min	Body_Max	Obj_Count	Act_Count	Attr_Count	AttrDisc_Count	AttrDisc_Body_Max
Gen01	10	1	5	5	1	5	10	100	6	1	5	1	5	10	1	5	100	3	3	1	5
Gen02	20	2	10	5	2	5	20	200	6	2	5	2	5	20	1	5	200	3	6	2	5
Gen03	30	4	15	5	3	5	30	300	6	3	5	4	5	30	1	5	300	3	9	4	5
Gen04	40	8	20	5	4	5	40	400	6	4	5	8	5	40	1	5	400	3	12	8	5
Gen05	50	16	25	5	5	5	50	500	6	5	5	16	5	50	1	5	500	3	15	16	5
Gen06	100	1	50	5	10	5	100	1000	6	10	5	1	5	100	1	5	1000	3	30	1	5
Gen07	200	2	100	5	20	5	200	2000	6	20	5	2	5	200	1	5	2000	3	60	2	5
Gen08	300	4	150	5	30	5	300	3000	6	30	5	4	5	300	1	5	3000	3	90	4	5
Gen09	400	8	200	5	40	5	400	4000	6	40	5	8	5	400	1	5	4000	3	120	8	5
Gen10	500	16	250	5	50	5	500	5000	6	50	5	16	5	500	1	5	5000	3	150	16	5
Gen11	1000	1	500	5	100	5	1000	10000	6	100	5	1	5	1000	1	5	10000	3	300	1	5
Gen12	2000	2	1000	5	200	5	2000	20000	6	200	5	2	5	2000	1	5	20000	3	600	2	5
Gen13	3000	4	1500	5	300	5	3000	30000	6	300	5	4	5	3000	1	5	30000	3	900	4	5
Gen14	4000	8	2000	5	400	5	4000	40000	6	400	5	8	5	4000	1	5	40000	3	1200	8	5
Gen15	5000	16	2500	5	500	5	5000	50000	6	500	5	16	5	5000	1	5	50000	3	1500	16	5
Gen16	6000	1	3000	5	600	5	10000	100000	6	1000	5	1	5	10000	1	5	100000	3	3000	1	5
Gen17	7000	2	3500	5	700	5	20000	200000	6	2000	5	2	5	20000	1	5	200000	3	6000	2	5
Gen18	8000	4	4000	5	800	5	30000	300000	6	3000	5	4	5	30000	1	5	300000	3	9000	4	5
Gen19	9000	8	4500	5	900	5	40000	400000	6	4000	5	8	5	40000	1	5	400000	3	12000	8	5
Gen20	10000	16	5000	5	1000	5	50000	500000	6	5000	5	16	5	50000	1	5	500000	3	15000	16	5

D.2 Negotiation Experiment Results

Experiment	Time (s)	Rounds
Gen01SB01	0.44	6
Gen02SB01	0.62	8
Gen03SB01	0.83	6
Gen04SB01	0.90	9
Gen05SB01	1.04	8
Gen06SB01	1.42	8
Gen07SB01	3.73	8
Gen08SB01	6.64	9
Gen09SB01	9.14	8
Gen10SB01	13.06	8
Gen11SB01	21.22	6
Gen12SB01	94.88	8
Gen13SB01	392.84	8
Gen14SB01	585.72	8
Gen15SB01	1030.49	8
Gen16SB01	1186.79	8
Gen17SB01	3828.00	6
Gen18SB01	17253.60	10
Gen01AB01	0.22	3
Gen02AB01	0.25	3
Gen03AB01	0.40	3
Gen04AB01	0.28	3
Gen05AB01	0.39	3
Gen06AB01	0.60	3
Gen07AB01	0.90	3
Gen08AB01	1.19	3
Gen09AB01	1.82	3
Gen10AB01	2.91	3
Gen11AB01	4.12	3
Gen12AB01	10.62	3
Gen13AB01	40.51	3
Gen14AB01	59.33	3
Gen15AB01	113.70	3
Gen16AB01	132.26	3
Gen17AB01	536.02	3
Gen18AB01	1502.38	3

D.3 Update Experiment Results

Experiment	Π	**UR** O	**UR** U	**UP**	**UC**	**CPU Time (s)**
Gen01	15	1	0	61	2	0.13
Gen02	37	1	0	149	3	0.17
Gen03	89	1	0	357	1	0.26
Gen04	74	1	0	297	1	0.22
Gen05	137	1	0	549	1	0.37
Gen06	177	1	0	709	1	0.59
Gen07	401	1	0	1605	2	2.72
Gen08	625	1	0	2501	2	7.47
Gen09	977	1	0	3909	1	22.57
Gen10	1263	1	0	5053	1	45.80
Gen11	1935	1	0	7741	2	198.20
Gen12	3748	1	0	14993	1	1660.50
Gen13	8104	1	0	32417	1	41678.31
Gen14	10868	1	0	43473	N/A	N/A

Appendix E

Old EBNF

Query	=	NAME ‘asks’ NAME · ACT · OBJ · PU‘;’
Policy	=	{NAME ‘says’ (Rule $\mid$ Definition) ‘;’}
Rule	=	Head [‘if’ Body]
Definition	=	Def-Obli $\mid$ Def-RelC $\mid$ Def-Desc
Head	=	Auth $\mid$ Attr ‘:’ SF · PIF $\mid$ Rel-Dir ‘:’ SF
Body	=	(BTerm $\mid$ Aggr $\mid$ Cons)[‘,’ Body]
BTerm	=	[‘not’] [PRIN ‘says’](Attr $\mid$ Desc $\mid$ Rel)
Auth	=	(‘allow’ $\mid$ ‘deny’) · PRIN · ACT · OBJ · PU · OBN
Attr	=	Prin · ATTR-NAME [{·Val}]
Def-Obli	=	‘define’ · ‘obligation’ · OBN · ACT · Prin
Def-RelC	=	‘define’ · ‘relchain’ · RCN · ‘(’Body‘)’
Def-Desc	=	‘define’ · ‘description’ · DN · VAR · ‘(’Body‘)’
Aggr	=	VAR ‘=’ Aggr-Op · VAR · ‘(’Body‘)’
	$\mid$	Aggr-Op · VAR · ‘(’Body‘)’ · Aggr-Cmp
Aggr-Cmp	=	(‘exactly’ $\mid$ ‘atleast’ $\mid$ ‘atmost’) · Val
	$\mid$	‘between’ · Val · Val
Aggr-Op	=	‘count’ $\mid$ ‘sum’ $\mid$ ‘min’ $\mid$ ‘max’
Desc	=	SUB · ‘description’ · DN
Rel	=	Rel-Dir $\mid$ Rel-Sind $\mid$ Rel-Rind
Rel-Dir	=	SUB · ‘relationship’ · REL-TYPE · SUB
Rel-Sind	=	SUB · ‘sindRelationship’ · RCN · SUB
Rel-Rind	=	SUB · ‘rindRelationship’ · NUM · SUB
Cons	=	Val (‘$<$’ $\mid$ ‘$>$’ $\mid$ ‘$\leq$’ $\mid$ ‘$\geq$’ $\mid$ ‘$=$’ $\mid$ ‘$\neq$’) Val
Prin	=	SUB $\mid$ OBJ
Val	=	NAME $\mid$ VAR $\mid$ NUM