Marcin Czenko

TuLiP

Reshaping Trust Management

Marcin Czenko

TuLiP - Reshaping Trust Management

TULIP
RESHAPING TRUST MANAGEMENT

Marcin Ryszard Czenko

Composition of the Graduation Committee:

Prof. Dr. S. Etalle	(Eindhoven University of Technology, University of Twente)
Prof. Dr. P.H. Hartel	(University of Twente)
Prof. Dr. Ir. M. Aksit	(University of Twente)
Prof. Dr. K.R. Apt	(CWI, Amsterdam)
Prof. Dr. W. Jonker	(Philips Research, University of Twente)
Dr. Ir. M.J. van Sinderen	(University of Twente)
Prof. Dr. S. De Capitani Di Vimercati	(University of Milan, Italy)
Dr. W.H. Winsborough	(University of Texas, San Antonio, USA)

Distributed and Embedded Security Group
P.O. Box 217, 7500 AE, Enschede, The Netherlands.

SenterNovem
Agency of the Dutch Ministry of Economic Affairs
P.O. Box 93144, 2509 AC The Hague, The Netherlands.

Freeband/I-Share: Sharing resources in virtual communities for storage, communications, and processing of multimedia data.
Funded by SenterNovem/BSIK project nr 3025.

CTIT PhD Thesis Series Number 09-148.
Centre for Telematics and Information Technology (CTIT)
P.O. Box 217, 7500 AE Enschede, The Netherlands.

IPA: 2009-16
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithms).

ISBN: 978-90-365-2854-2
ISSN: 1381-3617
DOI: 10.3990/1.9789036528542

Cover Design: Marcin and Beata Czenko.
Printed by Wöhrmann Print Service.

TULIP
RESHAPING TRUST MANAGEMENT

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, 26^{th} of June 2009 at 16:45.

by

Marcin Ryszard Czenko

born on 25 October 1978

in Szczecinek, Poland

This dissertation is approved by:

Prof. Dr. S. Etalle (promotor)
Prof. Dr. P.H. Hartel (promotor)

To my wife Beata, and my family.

Summary

In today's highly distributed and heterogeneous world of the Internet, sharing resources has become an everyday activity of every Internet user. We buy and sell goods over the Internet, share our holiday pictures using facebook™, "tube" our home videos on You Tube™, and exchange our interests and thoughts on blogs. We podcast, we are Linkedin™ to extend our professional network, we share files over P2P networks, and we seek advice on numerous on-line discussion groups. Although in most cases we want to reach the largest possible group of users, often we realise that some data should remain private or, at least, restricted to a carefully chosen audience. Access control is no longer the domain of computer security experts, but something we experience everyday.

In a typical access control scenario, the resource provider has full control over the protected resource. The resource provider decides who can access which resource and what action can be performed on this resource. The set of entities that can access a protected resource can be statically defined and is known *a priori* to the resource provider. Although still valid in many cases, such a scenario is too restrictive today. The resource owner is not only required, but often wants to reach the widest possible group of users, many of which remain anonymous to the resource provider. A more flexible approach to access control is needed.

Trust Management is a recent approach to access control in which the access control decision is based on *security credentials*. In a credential, the credential issuer states attributes (roles, properties) of the credential subject. For the credentials to have the same meaning across all the users, the credentials are written in a trust management language. A special algorithm, called a *compliance checker*, is then used to evaluate if the given set of credentials is compliant with the requested action on the requested protected resource. Finally, an important characteristic of trust management is that every entity may issue credentials.

In the original approach to trust management, the credentials are stored at a well-known location, so that the compliance checker knows where to search for the credentials. Another approach is to let the users store the credentials. Storing the credentials in a distributed way eliminates the single point of failure introduced by the centralised credential repository, but now the compliance checker must know where to find the credentials. Another difficulty of the distributed approach is that the design of a correct credential discovery algorithm comes at the cost of limiting the expressive power of the trust management language.

In this thesis we show that it is possible to build a generic, open-ended trust management system enjoying both a powerful syntax and supporting distributed credential storage. More specifically, we show how to build a trust management system that has:

- a formal yet expressive trust management language for specifying credentials,
- a compliance checker for determining if a given authorisation request can be granted given the set of credentials,
- support for distributed credential storage.

We call our trust management system TuLiP (Trust management based on Logic Programming).

In the thesis we also indicate how to deploy TuLiP in a distributed content management system (we use pictures as the content in our implementation). Using the same approach, TuLiP can improve existing P2P content sharing services by providing a personalised, scalable, and password-free access control method to the users. By decentralising the architecture, systems like facebook™ or You Tube™ could also benefit from TuLiP. By providing easy to use and scalable access control method, TuLiP can encourage sharing of private and copyrighted content under a uniform and familiar user interface. Also Internet stores, often deployed as a centralised system, can benefit from using the credential based trust management. Here, TuLiP can facilitate the business models in which the recommended clients and the clients of friendly businesses participate in customised customer rewarding programs (like receiving attractive discounts). By naturally supporting co-operation of autonomous entities using distributed credentials, we believe that TuLiP could make validation of business relationships easier, which, in turn, could stimulate creation of new business models.

Acknowledgements

This thesis would have never become a reality without an enormous help of my two promotors: Sandro Etalle and Pieter Hartel. I am grateful to Sandro Etalle for countless number of things. First of all, I thank Sandro for his commitment to my work. I cannot recall a single moment, when I had the feeling that the work I was doing was not important to him. Sandro's honesty was an important driving force that helped me to retain self-confidence and belief in that I am able to finish this Ph.D. thesis. Everything I learnt about what a good technical paper should look like, and how to approach the difficult task of writing a good scientific paper professionally comes from Sandro. I wish every Ph.D. student to have this level of commitment and this level of knowledge from the promotor as I had from Sandro Etalle. I also would like to thank Sandro and his wife Nicole for warm reception in Trento.

I would like to thank Pieter Hartel for giving me the opportunity of doing Ph.D. in his research group. Excellent working conditions, friendly atmosphere, and unbelievable level of flexibility in the working hours are the testimonies of Pieter's deep understanding for even highly eccentric Ph.D. student behaviour. I appreciate both Pieter's and his wife Marijke's concern to make us foreign students feel a bit more at home. I am especially thankful to Pieter for his involvement in improving both my written and spoken English. Regardless of the extent to which I met Pieter's expectations, I believe that my English is better than it was four years ago when I was starting my Ph.D. journey. Finally, I am impressed with Pieter's extra-natural ability to "process" hundreds of pages in only a few days and still being able to find not only spelling mistakes, but also technical inconsistencies that both me and Sandro overlooked. If there are any mistakes in this thesis, it is most certainly because I forgot to include some of the Pieter's comments.

I thank Jeroen Doumen, who was my daily advisor for majority of time. I thank him for all discussions we had: it was very important to me. In a special way I am grateful to Jan Kuper for his enthusiasm in helping me to understand basics of mathematical logic during the individual course he gave me. I am often returning to his visualisations of the darkest corners of deductive systems. I also thank Jerry den Hartog. It was always a pleasure (and often the necessity) to "use" his sharp brain. In particular, I thank Jerry that he spontaneously agreed to provide the Dutch translation of the summary of this thesis.

I would like to thank Mehmet Akşit, Krzysztof Apt, Willem Jonker, Marten van Sinderen, Sabrina De Capitani Di Vimercati, and Will Winsborough for accepting the invitation to be part of the graduation committee of this thesis. It is a great honour for me.

I thank the Freeband consortium for the funding for my research. In particular, I would like to acknowledge all the members of the I-Share project, especially Inald Lagendijk, Dick Epema, Johan Pouwelse, Arno Bakker, and Jan-David Mol.

I thank the Distributed and Embedded Security research group for being my scientific home for more than last four years. Thanks to Angelika, Ari, Damiano, Ileana, Gabriele, Ha, Jordan, Marnix, Michèl, Mohammed, Nikolay, Pierre, Qiwei, Ricardo, Richard, Yee Wei, and all the others that I do not mention here for having an enjoyable time together. In

particular, I would like to thank Ileana Buhan for all the feedback I received from her and especially for sharing with me her experiences with the publishing process of a Ph.D. thesis. I also much enjoyed the presence of the recent members of the DIES group: André, Ayse, Emmanuele, Giorgi, Luan, Mohsen, Qiang, Saeed, Svetla, Trajce, Wolter, and Zheng. Here, I am especially grateful to Ayse Morali, Saeed Sedghi, and Mohsen Saffarian for all the honest, friendly discussions we had. I also would like to thank our secretaries: Marlous, Nicole, and especially Nienke who was always there ready to help me. I have special regards to Fred Spiessens from the Security group at the Technical University of Eindhoven: I really enjoyed the discussions we had in Trondheim (and not only there).

I also appreciate the possibility to share my research results with the colleagues from the Security Ph.D. Association Netherlands (SPAN). In particular I would like to mention Jing Pan and Hugo Jonker with whom I had interesting (not only scientific) discussions.

I like to thank my Polish colleagues Anna Zych, Paweł Garbacki, Łukasz Chmielewski, Piotr Kordy, and Przemysław Pawelczak. The possibility to talk to someone with the same historical background was always a valuable intellectual refreshment. I am grateful to my Master Thesis supervisor, Dr. Jacek Wytrębowicz, who was an important step on my scientific path. Also my primary and high school teachers: Paweł Metkowski, Helena Mikołajczyk, Andrzej Falko, Waldemar Gręźlikowski, and Maria Siwula, all had large part in helping me believing in my own capabilities. I cannot forget about my friends from Poland: Adam Herda, Piotr Kleczyński and all members of the *Sterkom* team, and Wojtek Mielke. I am grateful to Father Rafał Golianek and Hubert Niewiadomski who agreed to be my *paranimphen*.

I owe special thanks to my family: especially to my father who died in 2000, my mother, and my two sisters. *Kim jestem dzisiaj, zawdzięczam również Wam.*

Finally, I thank my wife Beata for being with me all these days and for all her empathy, patience, and help.

At the very end, but with the highest respect, I thank God and all His Saints for being always with me: *In Te, Domine, sperávi.*

26 June 2009
Enschede, The Netherlands

Table of Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

An important aspect of computer security is access control. In a typical access control scenario there is one entity, the requester, which wants to access a protected resource, and a second entity, which is the resource provider. Typically, the resource provider decides if the request can be authorised or not based on the identity of the requester, its role, or attributes the requester possesses. The resource provider has full control over who can access which resource and what kind of action can be performed on that resource. The resource provider and the requester are the only entities involved in making the authorisation decision. In such a simple situation, the resource provider knows all the entities that want to access her protected resources and, even more importantly, the resource provider feels competent in assigning and evaluating the rights of each specific requester.

The simple scenario above does not apply to today's highly distributed and heterogeneous world of the Internet.The resource provider may be interested in providing her services to a broader group of users hitherto unknown or even anonymous to the resource provider. Also the requesters may not know in advance which resource providers are most appropriate to satisfy their needs.To be able to deal with different requesters coming from different security domains, we need a more generic and open-ended solution. The following example (we use the scenario presented in this example throughout the thesis) illustrates the transition from traditional access control to a more flexible solution brought to us by Trust Management:

Example 1.1 Electronic on-line store *eStore* gives 10 % discount to registered clients who are students of the University of Twente (UT). Alice and Bob are students and are registered clients of the *eStore*. The registration is a simple process. A student visits *eStore*, fills in a registration form and shows the student id card. An employee of *eStore* reviews the registration form and, if everything is in order, an appropriate student record is added to a database holding the registered customers. The database storing the registered students can then be used to decide who gets a discount and who does not (Fig. 1.1).
When a student of the UT visits the store on-line, in order to claim the discount, the student must prove her identity so that the store can check if the requester is a registered client who is also a student.
Imagine now that *eStore* wants to extend the offer by serving the students from other universities or even from universities in different countries. Now *eStore* has a problem as *eStore* does not feel competent to validate the student id coming from a university other than the UT. To solve the problem, *eStore* should ask some external entity to check if the client is a student.

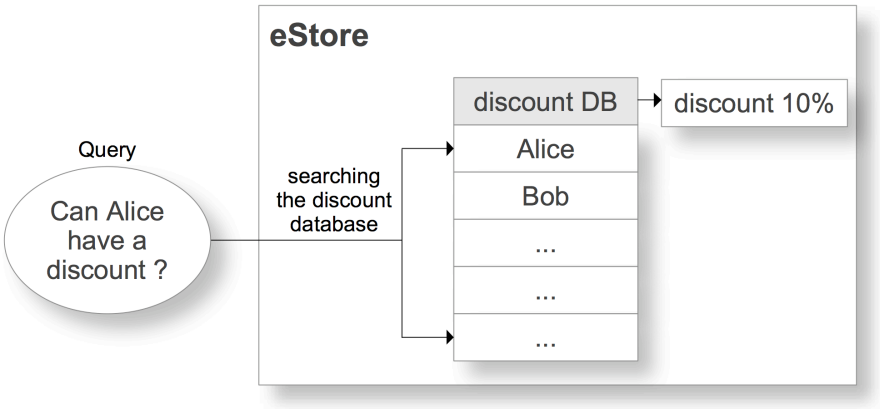


Fig. 1.1: The illustration of the scenario presented in Example 1.1

This would simplify the process for *eStore* by eliminating the need for complex registration procedure and would allow *eStore* to concentrate on the job *eStore* does the best: selling things.

Using the language of Trust Management, *eStore* would like to *delegate* the authority of deciding whether the requester is a student to some other entity which is more competent to make this decision. To accomplish this, in the Trust Management approach, *eStore issues* a *credential* in which *eStore* states which authorisation is being delegated and to whom. For instance, *eStore* may issue a credential in which *eStore* says that each student of the UT is eligible for a discount at *eStore*. The UT, in turn, for each student of the UT, may issue a credential in which the UT says that the given entity is a student of the UT. By evaluating the credentials, *eStore* may check if the requester should receive a discount at *eStore*. As we show it later in this section, by issuing additional credentials, *eStore* can handle students coming from other universities as well.

In Trust Management (TM), an access control decision is made by evaluating a set of *credentials*. An algorithm used for the evaluation of the credentials is called a *compliance checker*. The compliance checker is independent from the concrete implementation, which means that the result of the evaluation of the set of credentials should be the same regardless of the compliance checker being used (in other words, the proof is in the credentials not in the compliance checker). This is important because it allows the construction of a general purpose compliance checker which can be used in any application regardless of the actual security requirements.

Each entity has the right to issue a credential which can then be used by other entities to check compliance. The entity which issues the credential is called the credential issuer, and the entity receiving the authorisation defined in the credential is called the credential subject.

Example 1.2 Let us continue Example 1.1 and see how *eStore* can use Trust Management to build a more scalable service. Instead of storing the data of all registered students, *eStore*

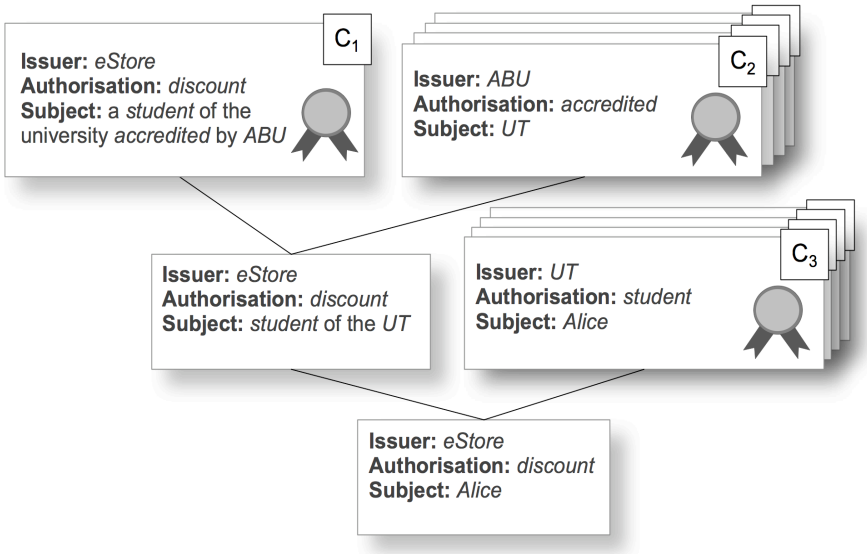


Fig. 1.2: The proof that Alice is eligible for a discount at the *eStore* (Example 1.2)

says that "any student of an accredited university has a discount". *eStore delegates* the authority of deciding which university is accredited to the *University Accreditation Board* (*ABU*). Similarly, *ABU* delegates the authority of deciding who is a student to every university that is accredited by *ABU*. Now eStore can check if an entity is a student by first checking if this entity is authorised by some university to use the role student and then by checking if this university is accredited by *ABU*. Figure 1.2 illustrates the process of proving that *Alice* is eligible for a discount at *eStore*. In the figure, we see the credentials being used in the proof (boxes labeled C_1, C_2, C_3) and the itermediate results constructed by the compliance checker during the processing (the two remaining boxes). Given credentials C_1 and C_2, the compliance checker can infer that a student of the *UT* can have a discount. Then, having also credential C_3, the compliance checker can conclude that *Alice* is a student of an accredited university and therefore can have a discount at the *eStore*.

The example above shows how trust management makes access control more manageable. Before, *eStore* had to keep track of the data of every registered student. Now *eStore* uses one credential in which *eStore* delegates the authorisation to *ABU* and, indirectly, to each accredited university. Similarly, *ABU* had to keep record only of the accredited universities, and each university must know only its own students. Everyone is doing its job.

Example 1.2 also shows the "trust" component of a trust management system. By delegating authority to another entity, *eStore trusts* that this entity will behave appropriately. In Example 1.2, *eStore trusts* that *ABU* accredits only real universities. Similarly, *eStore* and *ABU* trust accredited university to issue a student credential only to a real student. Enforcing the correct behaviour is a research problem on its own (known as a problem of the policy enforcement) and is outside the scope of this thesis. Instead, we focus on deciding what correct

behaviour is (i.e. compliance), which is a hard problem by itself.
Summarising, in the simplest form, a trust management system has the following two components:

1. A language (called a trust management language) for specifying credentials,

2. A compliance checker, which is a service for determining if a given authorisation request can be proven true given the set of credentials.

3. A policy enforcment scheme (beyond the scope of the thesis).

A language for specifying credentials should be expressive enough to satisfy the needs of different users while having a well-defined formal declarative semantics. The compliance checker should be proven sound and complete with respect to the declarative semantics of the trust management language yet should be easy to implement and use.

As mentioned above, security credentials can be issued by different entities. In the original idea of trust management [27, 28], it is assumed that the credentials are always available (i.e. all credentials are stored in some well-known location or there exists some mechanism that guarantees that the required credentials can be found). Li et al. [67] point out that credential distribution is a hard research problem on its own and they investigate possible storage options. Li et al. define a problem of a *credential chain discovery* which is that of finding a *chain* of credentials which proves that given request is true. Not being able to find a credential chain is the same as saying that there is no proof for the request, which means that the request cannot be authorised.

Example 1.3 Continuing Example 1.2, the most intuitive credential deployment scheme is when all credentials are stored by the credential issuer: credential (C_1) at *eStore*, credential (C_2) at *ABU*, and credential (C_3) at the *UT*. In this scenario all required credentials can be found simply by first fetching (C_1) from *eStore*, credential (C_2) form *ABU*, and credential (C_3) from the *UT*. This storage configuration has the disadvantage that one may need to query many accredited universities (i.e. TUD, TU/e, UvA, ...) before finding the university listing *Alice* as a student. Another possibility is to store credential (C_1) at *eStore*, credential (C_2) at *UT*, and credential (C_3) at *Alice*. Here, in constructing the proof of compliance, one can directly ask Alice if she is a student of some university and the request an "accreditation" credential from that university. Not every storage scheme, however, guarantees that the required credentials will be found. For instance, if credential (C_1) is stored at *eStore* and credentials (C_2) and (C_3) are both stored at the *UT*, then there is no way of finding credentials (C_2, C_3) and the query "is Alice eligible for a discount" would be answered negatively.

The list of the components a (distributed) trust management system should have becomes the following:

1. a trust management language,

2. a compliance checker,

3. support for the distributed credential storage.

Our requirements are not satisfied by the existing Trust Management systems, which can be divided into two groups. In the first group we have trust management systems like PeerTrust [74], PeerAccess [102], or $\mathcal{X}$-TNL [21] that provide an expressive logic-based trust management language, but which cannot be easily used in practice because of the complex syntax, no or limitted support for the credential distribution and discovery, and non-existing implementation. In the second group we have the family of trust management languages RT, which enjoys intuitive syntax and relatively simple semantics (at least for the simplest members of the family), supports credential distribution and discovery, but does not scale well when more sophisticated scenarios need to be modeled. Our goal is to reshape the credential based trust management world by delivering a system which combines the simplicity, expressive power, and extensibility of Prolog with the flexibility of distributed credential storage offered by RT.

Therefore we can formulate our research question as follows:

Can we build a generic open-ended trust management system enjoying powerful syntax and supporting distributed credential storage ?

In order to answer our research question we need to satisfy the following objectives:

To build a trust management system which: 1. supports flexible policies, and which provides a simple language to express the policies, 2. provides a flexible method for credential distribution and discovery.	**Theory**
To implement a compliance checker for such a trust management system such that this compliance checker, on one hand, is sound and complete w.r.t. the given semantics of the policy language and, on the other hand, can be readily deployed on a distributed system.	**Practice**

Flexible Language (Theory) The trust management system should be flexible enough to support various needs of the users. In particular, the language for specifying credentials should allow for expressing the most common access control policies (like threshold or separation of duty). A trust management language should also be scalable and extensible so that new (future) policies can be expressed easily. For instance, the language should accommodate the extensions allowing the user to express not only credential based but also reputation

based trust management policies. Finally, the trust management language should have a formal declarative semantics so that the intended meaning of a credential is independent from the particular realisation of the compliance checker.

Credential Distribution and Discovery (Theory) The trust management system should provide a well-defined mechanism for planning the credential distribution. The system should guarantee the consistence of storage so that the credentials can be found later. Finally, the information about the credential storage should be part of the declarative semantics of the trust management language so that the meaning of the credentials is idependent from the concrete implementation of the compliance checker.

Concrete Implementation (Practice) The trust management system should be relatively easy to deploy. By this we mean that a trust management system should not have heavy requirements on the underlying infrastructure like the need for a complex public key infrastructure (PKI) or third party authorities. Ideally, it should be possible to build the system using of-the-shelf components.

Li et al. [66, 67] show that a system which satisfies some of these requirements can be built in the form of the RT family of Trust Management languages. They present a storage type system to handle the credential distribution and discovery and they propose the credential chain discovery algorithm. The algorithm is proven sound and complete. One inconvenience in the RT framework is that in order to handle more complex access control policies, one needs to resort to different dialects of the language. Additionally, the storage information is not reflected in the language itself, which makes the storage type system less scalable (in fact the storage type system is provided only for the simplest member of the RT family: RT_0). In this thesis we show that one can design a trust management system having an expressive language, credential discovery algorithm and a storage type system all in one framework enjoying formal logic based reading.

1.1 Plan of the thesis

Figure 1.3 shows the roadmap of the thesis. We start in Chapter 2 with an introduction to trust management by presenting the RT family of trust management languages. In Chapter 2 we also present the problem of credential chain discovery and we show how RT solves this problem. Chapter 2 also presents the extensive related work on trust management and we discuss the relationship between trust management and reputation systems. The contents of this chapter was first published as M. R. Czenko, S. Etalle, D. Li, and W. H. Winsborough: *An Introduction to the Role Based Trust Management Framework RT*. In *Foundations of Security Analysis and Design IV – FOSAD 2006/2007 Tutorial Lectures*, volume 4677 of *LNCS*, pages 246–281. Springer Verlag, 2007.

In Chapter 3, we extend the RT family with a new member which is able to deal with non-monotonic policies. The new member is called $\mathrm{RT}_{\ominus}$ and introduces a restricted form of negation, called *negation in context*, by the means of the new operator $\ominus$. The contents of this chapter was first published as M. Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. den Hartog: *Nonmonotonic Trust Management for P2P Applications*. In *Proc. 1st*

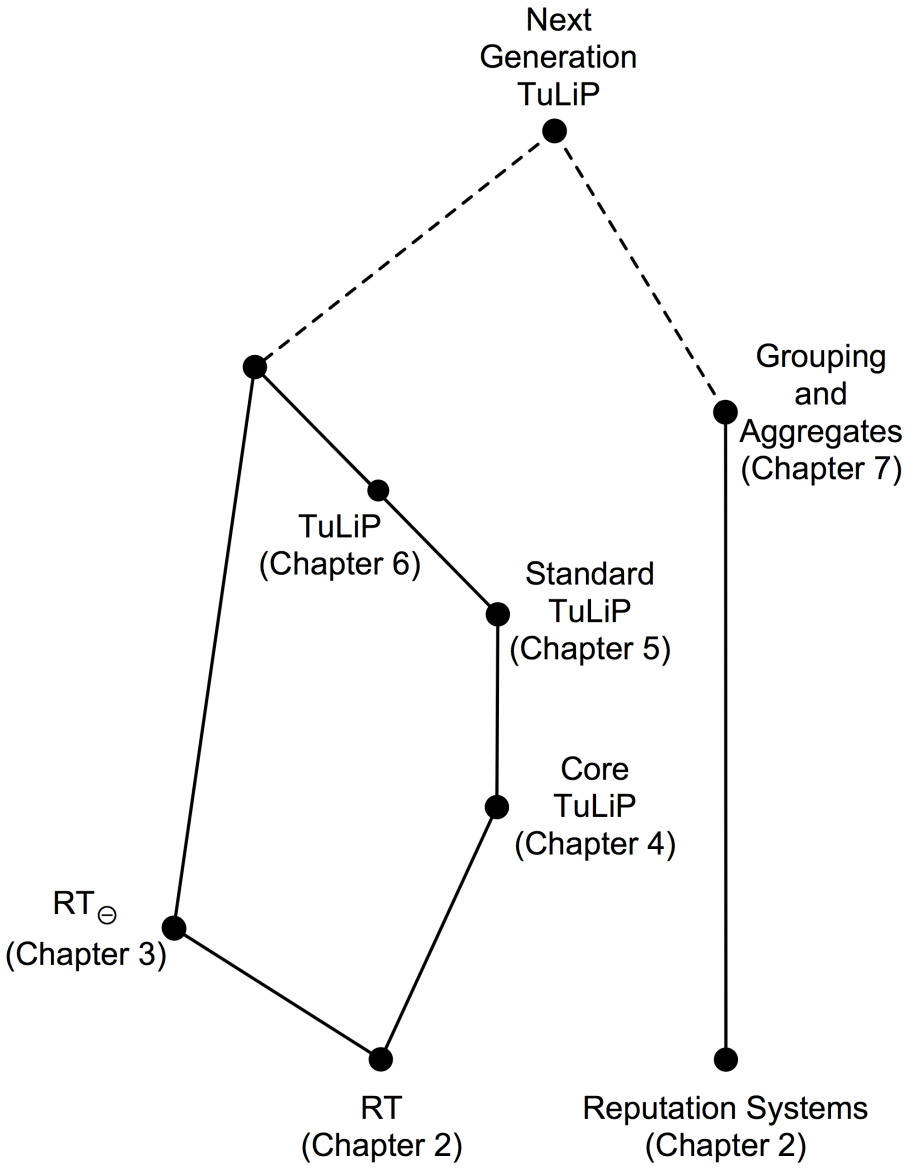


Fig. 1.3: The roadmap of the thesis

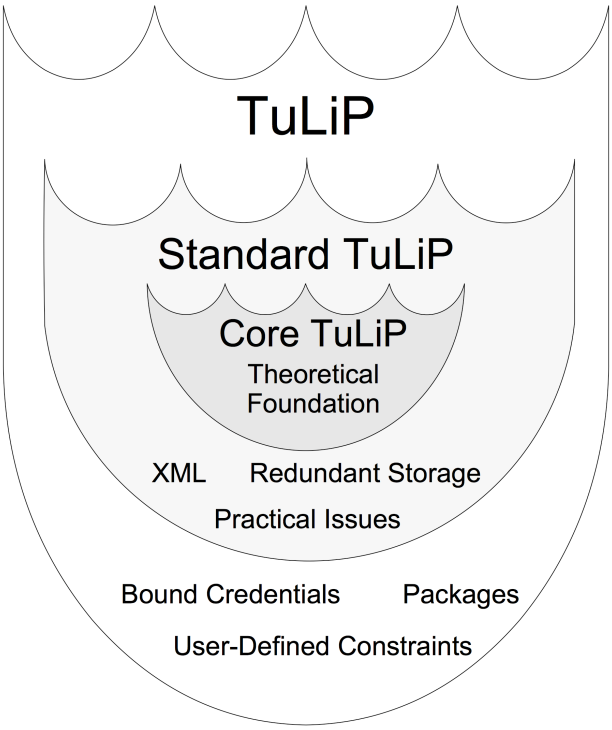


Fig. 1.4: From Core TuLiP to TuLiP

International Workshop on Security and Trust Management, Electronic Notes in Theoretical Computer Science (ENTCS), pages 101–116. Elsevier, 2005.

Having analysed the strengths and weaknesses of RT, in Chapter 4 we introduce the theoretical foundation for our own trust management system: *Core TuLiP*. In Core TuLiP we propose a new logic-based trust management language with uniform syntax and strong logic-based semantics. We show how to guide the credential distribution by using the mode system, which is a uniform mechanism indicating which entity should store the credential. Our mode system is the integral part of each TuLiP system where, in contrast, the type system in the RT family is only available for the simplest member of the family RT_0. Finally, in Chapter 4, we present the Lookup and Inference AlgoRthm (LIAR) which performs compliance checking and also discovers and fetches the missing credentials. We prove that LIAR is sound and complete w.r.t. the declarative semantics of our trust management language. The contents of this chapter was first published as M. R. Czenko and S. Etalle: *Core TuLiP - Logic Programming for Trust Management*. In *Proc. 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal*, volume 4670 of *LNCS*, pages 380–394, Berlin, 2007. Springer Verlag.

In Chapter 5, we present Standard TuLiP - a practical realisation of a trust manage-

ment system based on Core TuLiP. In this chapter we answer typical questions one needs to answer when deploying a new trust management system like how to encode the credentials for the efficient exchange and evaluation or what the minimal requirements are on the underlying infrastructure. We also describe a simple distributed contents sharing system using Standard TuLiP. A short presentation of our demonstration system can be found at *http://dies.cs.utwente.nl/~czenkom/tulip/doc*. An example of the mode set register can be found at *http://tulip.webphoto.nl*. The contents of this chapter was first published as M. R. Czenko, J. M. Doumen, and S. Etalle: *Trust Management in P2P Systems Using Standard TuLiP*. In *Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security*, Trondheim, Norway, volume 263/2008 of IFIP International Federation for Information Processing, pages 1-16, Boston, May 2008. Springer.

After showing that it is possible to implement a trust management language built on the theory given by Core TuLiP, in Chapter 6 we present a full TuLiP trust management system. Full TuLiP has solid theoretical foundation of Core TuLiP and formalises the concepts introduced informally when designing Standard TuLiP. Full TuLiP consists of an expressive trust management language supporting XML content and user-defined constraints, optimised and extended LIAR algorithm, and a storage type system supporting redundant storage of the credentials and the constraints.

In the last chapter, Chapter 7 we are going beyond the original research question. We show the theoretical foundation of grouping and aggregation in logic programming which can be used in the next generation of the TuLiP trust management system. Grouping and aggregation are the must-have features of a trust management language if one wants to bridge the credential based and reputation based trust management in one comprehensive system.

Figure 1.4 illustrates the features provided incrementally by different versions of the TuLiP trust management system.

1.2 Contributions

Our contributions can be summarised as follows:

1. We design a trust management language which allows us to write access control policies that depend on the access control policies of other users.

2. We show how to guide the credential distribution and discovery using a mode system.

3. We design a Lookup and Inference AlgoRithm (LIAR), which is our version of the compliance checker. LIAR answers the authorisation requests by discovering the credentials and building the proof for the given request at the same time. We prove the soundness and completeness of the algorithm w.r.t. the standard logic programming semantics.

4. We provide a proof of concept implementation of the TuLiP trust management system.

5. We provide the theoretical foundation for merging credential based and reputation based trust management.

In this thesis we answer our research question positively. Although the integration with the reputation based trust management is not yet fully addressed, we show in Chapter 7 that, in principle, this can be accomplished.

CHAPTER 2

An Introduction to the Role Based Trust Management Framework RT

In this chapter we set the context for the whole thesis by presenting one of the most successful credential-based trust management systems: RT. RT is a family of Role Based Trust Management (TM) languages with RT_0 being its simplest member. RT_0 is a simple yet powerful trust management language which allows us to capture the intuition behind credential-based trust management and to familiarise the reader with the challenges all trust management systems face. In the chapter we present a detailed syntax and semantics of RT_0 and we show how RT_0 deals with credential distribution and discovery by presenting the storage type system and the algorithms for the credential discovery.

We also present other members of the RT family by showing examples involving different dialects of RT. By showing a broad range of examples we hope the reader will understand of the expressive power a trust management system should provide to satisfy diverse requirements. On the other hand, we want to show that RT, although being successful in achieving its goals, leaves space for improvement in terms of flexibility of the syntax and ease of use. Here we also introduce our extension to RT, $RT_{\ominus}$, which provides a carefully controlled form of non-monotonicity. A formal treatment of $RT_{\ominus}$ is the subject of the next chapter of this thesis.

Finally, we discuss related work on trust management and the relationship between trust management and reputation systems.

The contents of this chapter was first published as M. R. Czenko, S. Etalle, D. Li, and W. H. Winsborough. *An Introduction to the Role Based Trust Management Framework RT*. In *Foundations of Security Analysis and Design IV – FOSAD 2006/2007 Tutorial Lectures*, volume 4677 of *LNCS*, pages 246–281. Springer Verlag, 2007.

2.1 Introduction

The problem of guaranteeing that confidential data is not disclosed to unauthorised users is paramount in our IT-dominated world. This problem is usually solved by implementing access control techniques. Traditional access control schemes make authorisation decisions based on the identity, or the role of the requester of a protected resource or service. However, in decentralised environments, the resource owner and the requester often are unknown to one another, making access control based on identity ineffective. To give a simple example, consider the situation in which a bookstore adopts the policy of giving a 10% discount to students of accredited universities. Although a certificate authority may assert that the name of the requester is Alice Q. Smith, if this name is unknown to the bookstore, the name itself does not aid in making a decision whether he is entitled to a discount or not. What is needed is information about the rights, qualifications, and other characteristics assigned to Alice Q. Smith by one or more authorities (in our example, the university he attends), as well as trust information about the authority itself (e.g. is it accredited?).

Trust management [25, 27, 44, 53, 57, 66, 87, 100] is an approach to access control in decentralised distributed systems with access control decisions based on policy statements made by multiple principals. In trust management systems, statements that are maintained in a distributed manner are often digitally signed to ensure their authenticity and integrity; such statements are called credentials or certificates. A key aspect of trust management is delegation: a principal may transfer limited authority over one or more resources to other principals.

RT [65, 66, 67] is a family of Role Based Trust Management languages introduced by Li, Winsborough and Mitchell. At its most abstract, the notion of role used is simply a set of principals. The primary application of RT is intended to be authorisation and access control, and the main purpose of roles is to confer to their members access to specific resources. Nevertheless, roles can also be used in a more general way. For instance, membership in the role of student at the University of Texas may entail certain privileges, but serves to characterise the status of its members more generally. Such characterisations facilitate granting new privileges to entire classes of users.

This chapter is meant as an introduction to the RT family of trust management languages. It contains a thorough description of RT_0 which is the core language of the family, and some examples of the more sophisticated members: RT_1, RT_2, RT^T, RT^D [66], and the later $RT_{\ominus}$, described in Chapter 3. Concerning RT_0, this chapter describes in detail, syntax, semantics, decentralised storage and credential chain discovery. Technically, the content of this chapter derives directly from the original papers [66, 67], with some changes which simplify the exposition while maintaining generality: in particular we employ a restriction on queries that simplifies the definition of credential graph (see Remark 2.3.1). We also introduce new pseudo-code versions of the credential chain discovery algorithms that we believe to be clearer than the originals.

The chapter is structured as follows: in Sect. 2.2 we present the syntax and the semantics of RT_0- the core member of the RT family. We also show several examples showing possible application areas for RT_0. In Sect. 2.3 we present the credential chain discovery algorithm. Here we define the backward, forward, and bidirectional search algorithms. Section 2.4 presents the storage type system for RT_0 and Sect. 2.5 shows by example other members of the RT family and their expressive power. Finally in Sect. 2.6 we present the related work

and in Sect. 2.7 we give conclusions.

2.2 RT_0

The RT framework encompasses a number of languages which have the same basic structure, while offering different features. The main members of the RT family are RT_0, RT_1, RT^T, and RT^D. Here we focus on the core member of RT: RT_0. Later, in Sect. 2.5, we present examples of the features of RT_1, RT^T, and RT^D.

2.2.1 Syntax

The basic constructs of RT_0 are *entities*, *role names* and *roles*. An *Entity* is also often called a principal, and can be a computer agent or an individual. An entity can define roles, issue credentials, and make requests. An entity can define roles, issue credentials, and make requests. In general, an entity may be identified by a public key, or by a user account; following Li et al. [67], we abstract away from the mechanism used for identifying entities. We denote an entity by a name starting with an uppercase letter (possibly with a subscript), e.g. A, B, B_1, and *Alice* are all entities. A *role name*, on the other hand, is denoted by a string starting with a lowercase letter (possibly with a subscript), like r, r_1, and *student*.

Finally, a *role* has the form of an entity followed by a role name, separated by a dot. For example, $A.r$, $B.r_1$, and *University.student* are valid roles. The notion of a role is central to RT_0. A role $A.r$ denotes the set of entities that are members of this role – a set that we refer to informally by *members*$(A.r)$. A is called the *owner* of the role $A.r$, and is the only authority that can directly determine which are the members of $A.r$.

A permission in RT_0 is represented by a role. For example, the permission to read a confidential document on a corporate network of a company C can be represented by role *C.readConfidential*: in this case, an entity has read permission if and only if the entity belongs to *members*(*C.readConfidential*). Other roles are used to represent other properties, sometimes called *attributes*, that characterise the members or their relationship to the role owner. For example, membership in *C.employee* might indicate an employment relationship with C. This example illustrates one aspect of how RT supports decentralisation by making the entity with which one has an employment relationship explicit. In RT there is no notion of simply being employed without mentioning the entity whose judgement is being asserted or whose consent makes it so.

There are four types of credentials in RT_0 that an entity A can issue, each corresponding to a different way of defining the membership of $A.r$:

- *Simple Member*: $A.r \longleftarrow D$.

  With this credential A asserts that D is a member of $A.r$.

- *Simple Inclusion*: $A.r \longleftarrow B.r_1$.

  With this credential A asserts that $A.r$ includes (all members of) $B.r_1$. This represents a delegation from A to B, as B may cause new entities to become members of the role $A.r$ by issuing credentials defining (and extending) $B.r_1$.

- *Linking Inclusion*: $A.r \longleftarrow A.r_1.r_2$.

  $A.r_1.r_2$ is called a *linked role*. With this credential A asserts that $A.r$ includes $B.r_2$ for every B that is a member of $A.r_1$. This represents a delegation from A to all the members of the role $A.r_1$.

- *Intersection Inclusion*: $A.r \longleftarrow B_1.r_1 \cap B_2.r_2$.

  $B_1.r_1 \cap B_2.r_2$ is called an *intersection*. With this credential A asserts that $A.r$ includes every principal who is a member of both $B_1.r_1$ and $B_2.r_2$. This represents partial delegation from A to B_1 and to B_2.

In the original paper introducing RT_0 [67], the number of intersection elements in the intersection inclusion credentials is unlimited. Also, each intersection element can be either a role or a linked role. Here we restrict the number of intersection elements to two and require that each intersection element be a role. This makes the description easier to follow and simplifies some definitions. However it imposes no restriction on the expressive power of the language. A credential of the more general form can be replaced by several of the more restricted credentials presented above by introducing auxiliary roles, splitting longer intersections into several intersection inclusions, and introducing a linking inclusion for each linked role.

A *policy* is a finite set of credentials. We use the term *role expression* for any entity, role, linked role, or intersection; thus each RT_0 credential has the form $A.r \longleftarrow e$, where e is a role expression. Such a credential means that *members*$(e) \subseteq$ *members*$(A.r)$. We say that this credential *defines* the role $A.r$. Further, we call A the *issuer*, e the *body* and each entity occurring syntactically in e a *subject* of this credential. To be precise, the set *base*(e) of subjects of $A.r \longleftarrow e$ is defined as follows: *base*$(A) = \{A\}$, *base*$(A.r) = \{A\}$, *base*$(A.r_1.r_2) = \{A\}$, and $base(B_1.r_1 \cap B_2.r_2) =$ *base*$(B_1.r_1) \cup$ *base*$(B_2.r_2) = \{B_1, B_2\}$.

2.2.2 Semantics

In this section, we present the declarative semantics of RT_0. We follow Li et al. [66] and do this in terms of the semantics for logic programs by providing a translation of a policy $\mathcal{C}$ to a Datalog program, which we call the *semantic program*. The set-theoretic semantics for RT_0 can be found in Li et al. [67].

Given a set $\mathcal{C}$ of RT_0 credentials (i.e. a policy) the corresponding *semantic program*, $SP(\mathcal{C})$, is a Datalog program with one ternary predicate m. Intuitively, $m(A, r, D)$ indicates that D is a member of the role $A.r$. Given an RT statement $c \in \mathcal{C}$, the *semantic program* of c, $SP(c)$, is defined as follows (identifiers starting with "?" are logical variables):

$$\begin{aligned}
SP(A.r \longleftarrow D) &= m(A, r, D). \\
SP(A.r \longleftarrow B.r_1) &= m(A, r, ?X) :- m(B, r_1, ?X). \\
SP(A.r \longleftarrow A.r_1.r_2) &= m(A, r, ?X) :- m(A, r_1, ?Y), m(?Y, r_2, ?X). \\
SP(A.r \longleftarrow B_1.r_1 \cap B_2.r_2) &= m(A, r, ?X) :- m(B_1, r_1, ?X), m(B_2, r_2, ?X).
\end{aligned}$$

SP extends to a set of statements as expected: $SP(\mathcal{C}) = \{SP(c) \mid c \in \mathcal{C}\}$. Finally, given a policy $\mathcal{C}$, the semantics of a role $A.r \in \mathcal{C}$ is defined in terms of atoms entailed by the semantic program.

Definition 2.2.1 (Semantics of a Role) *Let $\mathcal{C}$ be an RT_0 policy, and let $SP(\mathcal{C})$ be the corresponding semantic program. The semantics of a role is defined as follows:*

$$[\![A.r]\!]_{SP(\mathcal{C})} = \{D \mid SP(\mathcal{C}) \models m(A, r, D)\}.$$

2.2.3 Examples

We now present some examples presenting how RT_0 can be used in different application areas. We begin with an example from Li et al. [67], showing a typical scenario from the area of electronic commerce.

Example 2.1 $EPub$ is an electronic publishing company that offers a special discount to anyone who is both a preferred customer of the sister organisation, *EOrg*, and an *ACM* member. *Alice* is both. We have the following set $\mathcal{C}$ of credentials:

(1) *EPub.spdiscount* $\longleftarrow$ *EOrg.preferred* $\cap$ *ACM.member*
(2) *EOrg.preferred* $\longleftarrow$ *EOrg.university.student*
(3) *EOrg.university* $\longleftarrow$ *ABU.accredited*
(4) *ABU.accredited* $\longleftarrow$ *StateU*
(5) *StateU.student* $\longleftarrow$ *RegistrarB.student*
(6) *RegistrarB.student* $\longleftarrow$ *Alice*
(7) *ACM.member* $\longleftarrow$ *Alice*
(8) *ACM.member* $\longleftarrow$ *Bob*

The semantic program, $SP(\mathcal{C})$, corresponding to the above policy is:

(1) $m(\textit{EPub}, \textit{spdiscount}, ?X) :- m(\textit{EOrg}, \textit{preferred}, ?X), m(\textit{ACM}, \textit{member}, ?X).$
(2) $m(\textit{EOrg}, \textit{preferred}, ?X) :- m(\textit{EOrg}, \textit{university}, ?Y), m(?Y, \textit{student}, ?X).$
(3) $m(\textit{EOrg}, \textit{university}, ?X) :- m(\textit{ABU}, \textit{accredited}, ?X).$
(4) $m(\textit{ABU}, \textit{accredited}, \textit{StateU}).$
(5) $m(\textit{StateU}, \textit{student}, ?X) :- m(\textit{RegistrarB}, \textit{student}, ?X).$
(6) $m(\textit{RegistrarB}, \textit{student}, \textit{Alice}).$
(7) $m(\textit{ACM}, \textit{member}, \textit{Alice}).$
(8) $m(\textit{ACM}, \textit{member}, \textit{Bob}).$

The semantics of the roles defined by the set of credentials above is then the following:

$[\![\textit{EPub.spdiscount}]\!]_{SP(\mathcal{C})} = \{\textit{Alice}\}$
$[\![\textit{EOrg.preferred}]\!]_{SP(\mathcal{C})} = \{\textit{Alice}\}$
$[\![\textit{ACM.member}]\!]_{SP(\mathcal{C})} = \{\textit{Alice, Bob}\}$
$[\![\textit{EOrg.university}]\!]_{SP(\mathcal{C})} = \{\textit{StateU}\}$
$[\![\textit{ABU.accredited}]\!]_{SP(\mathcal{C})} = \{\textit{StateU}\}$
$[\![\textit{StateU.student}]\!]_{SP(\mathcal{C})} = \{\textit{Alice}\}$
$[\![\textit{RegistrarB.student}]\!]_{SP(\mathcal{C})} = \{\textit{Alice}\}$

We see then that only *Alice* is eligible for a discount as *Bob*, though being a member of *ACM*, is not a student of an accredited university.

This example shows the basic use of the delegation (simple inclusion) and also how the linking inclusion can be used to build a scalable policy. For instance, by adding new members to role *ABU.accredited* one can extend the number of beneficiaries of a discount offered by *EPub* without directly contacting *EPub* or *EOrg* and changing their credentials.

The next example presents the use of RT_0 in collaborating organisations. This example originally appeared in Etalle and Winsborough [46].

Example 2.2 Consider the situation in which two companies: *CITA* (in Italy) and *CUS* (in the US), work on a joint project. *CITA* and *CUS*, have different management structures:

$$\begin{array}{ll} CITA.partner \longleftarrow Antonio & CUS.ceo \longleftarrow Bob \\ CITA.manager \longleftarrow Luca & CUS.employee \longleftarrow John \\ CITA.programmer \longleftarrow Sandro & CUS.employee \longleftarrow David \\ CITA.all \longleftarrow CITA.partner & CUS.all \longleftarrow CUS.ceo \\ CITA.all \longleftarrow CITA.manager & CUS.all \longleftarrow CUS.employee \\ CITA.all \longleftarrow CITA.programmer & \end{array}$$

In both companies there is an agreement that employees may trust all the sources that are trusted by the *partner* (resp. *ceo*). They can – of course – trust other sources as well.

$$\begin{array}{ll} Luca.partner \longleftarrow CITA.partner & John.ceo \longleftarrow CUS.ceo \\ Luca.trusted \longleftarrow Luca.partner.trusted & John.trusted \longleftarrow John.ceo.trusted \\ Sandro.partner \longleftarrow CITA.partner & David.ceo \longleftarrow CUS.ceo \\ Sandro.trusted \longleftarrow Sandro.partner.trusted & David.trusted \longleftarrow David.ceo.trusted \end{array}$$

CITA and *CUS* decide to join forces on *projX*, and they agree that most of the documents developed in *projX* should be accessible only to people working on the project, and that some particularly confidential documents should circulate only among the senior personnel. To implement this, the two companies agree to employ the role names *projX* and *seniorprojX*. In *CITA*, the partner decides who participates in projectX, and decides (in agreement with *CUS*) that the managers of *CITA* should be considered senior people, while in *CUS*, the ceo delegates to *John* the definition of the projectX team as well as of the senior people in it. Finally, *CITA* and *CUS* trust each other's definitions of (senior) people working on projectX. This policy is described and implemented by the following set of credentials.

$$\begin{array}{l} CITA.projX \longleftarrow Antonio.projX \\ CITA.seniorprojX \longleftarrow CITA.partner \\ CITA.seniorprojX \longleftarrow CITA.projX \cap CITA.manager \\ Antonio.projX \longleftarrow Luca \\ Antonio.projX \longleftarrow Sandro \\ CITA.projX \longleftarrow CUS.projX \\ CITA.seniorprojX \longleftarrow CUS.seniorprojX \\ CUS.projX \longleftarrow John.projX \\ CUS.seniorprojX \longleftarrow CUS.ceo \end{array}$$

$CUS.seniorprojX \longleftarrow John.seniorprojX$
$John.seniorprojX \longleftarrow John$
$John.projX \longleftarrow John$
$John.projX \longleftarrow David$
$CUS.projX \longleftarrow CITA.projX$
$CUS.seniorprojX \longleftarrow CITA.seniorprojX$

Example 2.2 shows again that by the proper use of the delegation (simple inclusion) and the linking inclusion one can build a sophisticated policy handling complex hierarchical relationships in an organisation. The example demonstrates that complex policies can be effectively modeled and still are easy to manage and understand.

The following two examples were initially presented by Winsborough and Li in [101]. The first of them shows an example of a co-operation between banking institutions and universities when providing financial support for students. Then, we show an example of policies that can be used by medical suppliers and charity organisations when handling natural disasters.

Example 2.3 A bank wants to know whether an entity is a full time student in order to determine whether the entity is eligible to defer repayment on a guaranteed student loan (GSL). (The US government insures banks against default of GSLs and requires participating banks to allow full-time students to defer repayments.) The *StateU* university may define its full-time student attribute by the following two credentials:

$StateU.fullTimeStudent \longleftarrow RegistrarB.fullTimeStudent$
$StateU.fullTimeStudent \longleftarrow StateU.phdCandidate \cap RegistrarB.partTimeStudent$

We see that *StateU* says that one is a full-time student if either *RegistrarB* says so, or if one is registered as a Ph.D. candidate at *StateU* and considered part-time student by *RegistrarB*. The following credentials, together with the above ones, show that Bob is a full-time student, i.e. $Bob \in [\![StateU.fullTimeStudent]\!]_{SP(\mathcal{C})}$:

$StateU.phdCandidate \longleftarrow StateU.gradOfficer.phdCandidate$
$StateU.gradOfficer \longleftarrow Carol$
$Carol.phdCandidate \longleftarrow Bob$
$RegistrarB.partTimeStudent \longleftarrow Bob$

Now, assume that *StateU* is certified by accreditation board *ABU*.

$$ABU.accredited \longleftarrow StateU$$

If universities define *fullTimeStudent* appropriately (for example, as done by *StateU* above), *BankWon* can issue credentials like those below to grant loan-deferment permission (denoted by *BankWon.deferGSL*) to students like Bob.

$BankWon.deferGSL \longleftarrow BankWon.university.fullTimeStudent$
$BankWon.university \longleftarrow ABU.accredited$

One can check that $Bob \in [\![BankWon.deferGSL]\!]_{SP(\mathcal{C})}$.

Example 2.3 shows again how flexible is RT_0. Here, by using delegation

$$BankWon.university \longleftarrow ABU.accredited$$

and the linking inclusion

$$BankWon.deferGSL \longleftarrow BankWon.university.fullTimeStudent$$

any full time student of an accredited university can be granted a deferred GSL.

Example 2.4 In the aftermath of a large natural disaster, *MedSup*, a medical supply merchant, offers to sell at a discount medical supplies to be used in the official clean up, which is being organised by a coalition called *ReliefNet*. *Alice* works for *MedixFund*, one of several charity organisations that use private contributions to obtain emergency medical supplies for emergency teams working at the disaster site. The following four credentials show that *Alice* is authorised for the discount.

(1) $MedixFund.pA \longleftarrow Alice$
(2) $ReliefNet.coaMember \longleftarrow MedixFund$
(3) $MedSup.partner \longleftarrow ReliefNet.coaMember$
(4) $MedSup.discount \longleftarrow MedSup.partner.pA$

Prior to joining the coalition, *MedixFund* issued credential (1), which states that *Alice* is a purchasing agent for the fund. One of *ReliefNet*'s responsibilities is to identify coalition-member organisations, as it does in credential (2). *MedSup* recognises these organisations as its coalition partners, as in credential (3), and offers discounted sales to the purchasing agents of those partners, as stated in credential (4). In this example, the judgements of *MedixFund*, *ReliefNet*, and *MedSup* are combined to authorise *Alice*'s receiving a discount from *MedSup*.

In the example above, when *MedSup* enters into another coalition, it can add an additional credential defining *MedSup.partner* to give the discount to the purchasing agents of its new partners. This is possible again thanks to using the simple inclusion and linking inclusion credentials.

With the increasing popularity of the P2P networks and their excellent support for sharing of user generated content, a high demand for flexible user-oriented policies can be observed. Below, we show an example of how RT_0 facilitates the use of personal policies in a heterogeneous P2P environment.

Example 2.5 Charles wants to share his pictures using a P2P file sharing system. He gives access to his gallery to his friends, friends of his friends, friends of firends of his friends, etc. For his movie collection, Charles applies a somewhat stronger policy: to access it, one has to be a member of Charles's *friend* role, and a member of the film club Charles is also a member of. The set of credentials, $\mathcal{C}$, modelling this scenario is shown below:

$Charles.accessMovies \longleftarrow Charles.friend \cap Charles.filmClub$
$Charles.accessPictures \longleftarrow Charles.friend$
$Charles.friend \longleftarrow Charles.friend.friend$
$Charles.friend \longleftarrow Alice$

Charles.friend $\longleftarrow$ *Bob*
Charles.filmClub $\longleftarrow$ *Johan*
Alice.friend $\longleftarrow$ *Jeffrey*
Bob.friend $\longleftarrow$ *Johan*
Johan.friend $\longleftarrow$ *Sandro*

Example 2.5 emphasises the fact that the delegation depth in RT_0 is unlimited. In the example, Charles's role *friend* contains not only friends of his friends, but also friends of friends of his friends and so on (*friends* is a transitive closure of the set of Charles's friends). Therefore, for the given set of credentials, we have the following semantics:

$$[\![\textit{Charles.accessMovies}]\!]_{SP(\mathcal{C})} = \{\textit{Johan}\}$$
$$[\![\textit{Charles.accessPictures}]\!]_{SP(\mathcal{C})} = \{\textit{Alice, Bob, Jeffrey, Johan, Sandro}\}$$

2.3 RT_0: The Credential Chain Discovery Algorithm

We have seen how RT_0 can be used to define roles and how roles can represent permissions or attributes. We now illustrate the mechanisms needed to answer the *queries* in the RT system. To set the stage, let us first enumerate the three *sorts of queries* we need to cope with. Let $\mathcal{C}$ be a set of credentials.

Sort 1 Given a role $A.r$ and an entity D, determine whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.

Sort 2 Given a role $A.r$, determine its member set, $[\![A.r]\!]_{SP(\mathcal{C})}$.

Sort 3 Given an entity D, determine all the roles it is a member of, i.e. generate the set $\{A.r \mid D \in [\![A.r]\!]_{SP(\mathcal{C})}\}$.

Notice that while queries of Sort 1 simply require a yes/no answer, the other two sorts require to generate a whole set. Also, notice that queries of Sort 2 and 3 are strictly more expressive than queries of Sort 1: if we are able to answer a query of Sort 2 or 3 we are certainly able to answer a query of Sort 1, while the opposite is not true. At this stage, one might wonder if Sort 3 queries are actually needed. This will become clear in the sequel.

Remark 2.3.1 *Technically, this section is based on Li et al. [67] with the additional simplifying assumption that queries may refer only to roles and principals (and not to role expressions, e.g. we do not allow queries such as "given a role expression $A.r_1.r_2$, determine its member set $[\![A.r_1.r_2]\!]_{SP(\mathcal{C})}$"). This assumption allows us to simplify the notation by a great deal, and does not limit the expressiveness of the framework, as one can always introduce a new role to take the meaning of a role expression.*

The algorithms we present in this section operate on a *credential graph*, which is a directed graph representing a set $\mathcal{C}$ of credentials and is built as follows: each node $[e]$ represents a role expression e; every credential $A.r \longleftarrow e$ in $\mathcal{C}$ contributes to the graph an edge from $[e]$ (the node representing e) to $[A.r]$ (the node representing $A.r$), which is denoted by

$[A.r] \Leftarrow [e]$, and is called a credential edge. A path in the graph from the node $[e_1]$ to the node $[e_2]$ consists of zero or more edges and is denoted $[e_2] \overset{*}{\Leftarrow} [e_1]$. Additional edges, called *derived edges*, are added to handle linked roles and intersections. These edges are called derived edges because their inclusion in the credential graph comes from the existence of other, semantically related, paths in the graph.

Given a set $\mathcal{C}$ of credentials, we define the following finite structures: Entities($\mathcal{C}$) is the set of entities in $\mathcal{C}$, Names($\mathcal{C}$) is the set of role names in $\mathcal{C}$, and RoleExpressions($\mathcal{C}$) is the set of role expressions that can be constructed using Entities($\mathcal{C}$) and Names($\mathcal{C}$), i.e.:

$$\text{RoleExpressions}(\mathcal{C}) = \begin{cases} A, \\ A.r_1, & \text{where} \quad A, B_1, B_2 \in \text{Entities}(\mathcal{C}), \\ A.r_1.r_2, & \qquad\quad r_1, r_2 \in \text{Names}(\mathcal{C}) \\ B_1.r_1 \cap B_2.r_2 \end{cases}$$

The following definition is a simplified version of Definition 2 given by Li et al. [67] (see Remark 2.3.1). Thanks to this simplification we can restrict our attention to the *basic* credential graph and avoid some complexities from the original presentation.

Definition 2.3.2 (Basic Credential Graphs) *Let $\mathcal{C}$ be a set of RT$_0$ credentials. The* basic credential graph *$G_\mathcal{C}$ relative to $\mathcal{C}$ is defined as follows: the set of nodes $N_\mathcal{C}$ = RoleExpressions ($\mathcal{C}$) and the set of edges $E_\mathcal{C}$ is the least set of edges over $N_\mathcal{C}$ that satisfies the following three closure properties:*

- *Closure property 1: If $A.r \longleftarrow e \in \mathcal{C}$, then $[A.r] \Leftarrow [e] \in E_\mathcal{C}$. $[A.r] \Leftarrow [e]$ is called a* credential edge.
- *Closure property 2: If there exists a path $[A.r_1] \overset{*}{\Leftarrow} [B]$ in $G_\mathcal{C}$, then $\forall r_2 \in$ Names($\mathcal{C}$), $[A.r_1.r_2] \Leftarrow [B.r_2] \in E_\mathcal{C}$. We call $[A.r_1.r_2] \Leftarrow [B.r_2]$ a derived link edge, and $\{[A.r_1] \overset{*}{\Leftarrow} [B]\}$ is a* support set *for this edge.*
- *Closure property 3: If $D, B_1.r_1 \cap B_2.r_2 \in N_\mathcal{C}$, and there exist paths $[B_1.r_1] \overset{*}{\Leftarrow} [D]$ and $[B_2.r_2] \overset{*}{\Leftarrow} [D]$ in $G_\mathcal{C}$, then $[B_1.r_1 \cap B_2.r_2] \Leftarrow [D] \in E_\mathcal{C}$. This is called a* derived intersection edge*, and $\{[B_1.r_1] \overset{*}{\Leftarrow} [D], [B_2.r_2] \overset{*}{\Leftarrow} [D]\}$ is a* support set *for this edge.*

The set of edges $E_\mathcal{C}$ can be constructed inductively as follows. We start with the set $E^0_\mathcal{C} = \{[A.r] \Leftarrow [e] \mid A.r \longleftarrow e \in \mathcal{C}\}$ and then construct $E^{i+1}_\mathcal{C}$ from $E^i_\mathcal{C}$ by adding one edge according to either closure property 2 or 3. Since $N_\mathcal{C}$ is finite, the order in which edges are added is not important, and the sequence $\{E^i_\mathcal{C}\}_{i \in \mathcal{N}}$ converges to $E_\mathcal{C}$.

Example 2.6 *Figure 2.1 shows a subset of the basic credential graph for the set of credentials in Example 2.1. Edges labelled with numbers are credential edges, and the numbers correspond to the ones marking credentials in Example 2.1. The two edges without labels are derived edges: one added by the closure property 2 ($[EOrg.university.student] \Leftarrow [StateU.student]$), and one by the closure property 3 ($[EOrg.preferred \cap ACM.member] \Leftarrow [Alice]$).*

Li et al. [67] shows that the credential graphs are sound and complete w.r.t. to the set-theoretic semantics: if there is a path $[e_2] \overset{*}{\Leftarrow} [e_1]$ in any $G_\mathcal{C}$, then $expr[\mathcal{S}_\mathcal{C}](e_2) \supseteq$

$expr[\mathcal{S}_\mathcal{C}](e_1)$, and if $D \in expr[\mathcal{S}_\mathcal{C}](e_0)$, then there exists path $[e_0] \overset{*}{\Leftarrow} [D]$ in $G_\mathcal{C}$. Here $expr[\mathcal{S}_\mathcal{C}](e)$ is the set-theoretic semantics of a role expression e, which can be proven in a straightforward way to be equivalent to the LP based semantics we have introduced in Sect. 2.2.2.

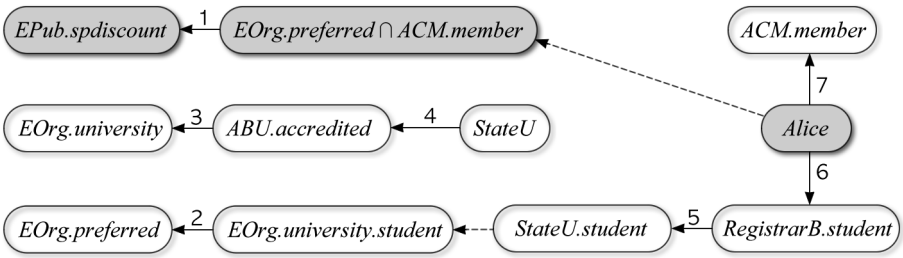


Fig. 2.1: The subset of the credential graph for the set of credentials in Example 2.1 containing path from *EPub.spdiscount* to *Alice*

Therefore, given a set $\mathcal{C}$ of credentials, we can answer each of the queries enumerated at the beginning of this section by consulting a basic credential graph of $\mathcal{C}$. Constructing the path $[A.r] \overset{*}{\Leftarrow} [D]$ alone proves that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$, provided that each derived edge has at least one support set. The portion of the credential graph that must be constructed for it is what we call a *credential chain*.

Definition 2.3.3 (Credential Chains) *Given a set $\mathcal{C}$ of credentials, a role $A.r$, and an entity D, a* credential chain *from D to $A.r$, denoted $\langle A.r \twoheadleftarrow D \rangle$, is a minimal subset of $E_\mathcal{C}$ containing a path $[A.r] \overset{*}{\Leftarrow} [D]$ and also containing a support set for each derived edge in the subset.*

The chain discovery starts at the node representing the requester, or at the node representing the role (permission) to be proven, or both, and then traversing paths in the graph trying to build an appropriate chain. In addition to being goal-directed, this approach allows the elaboration of the graph to be scheduled flexibly. Also, the graphical representation of the evaluation state makes it relatively straightforward to manage cyclic dependencies.

In the rest of this section we illustrate the three algorithms originally defined by Li et al. [67] to answer the three sorts of queries, listed at the top of this section (with the simplifying assumption illustrated in Remark 2.3.1). The backward search algorithm (also called the top-down algorithm) (Sect. 2.3.1) answers the second sort of queries, i.e. it determines all members of a role expression. The forward search algorithm (also called the bottom-up algorithm) in Sect. 2.3.2 answers the third sort of queries, i.e. it determines all roles that an entity is a member of. The bidirectional search algorithm (Sect. 2.3.3) answers the first sort of queries, i.e. it determines whether an entity is a member of a role expression. Note that in this section we assume that credentials are stored in such a way that we can list them all at any time. In practice, this is not always the case. We address the problem of distributed storage in the next section.

2.3.1 The Backward Search Algorithm

The backward search algorithm can determine all the members of a given role $A.r$. In terms of the credential graph, the algorithm finds all the entity nodes that can reach the node $A.r$, and for each such entity D, the algorithm constructs a chain $\langle A.r \twoheadleftarrow D \rangle$. The algorithm is called backward because it follows edges in the reverse direction. The algorithm works by constructing a *proof graph*, which is a data structure that represents a credential graph and maintains certain information on the nodes. Listing 2.1 shows the algorithm in pseudo-code using a Python-like syntax. We have four classes: *ProofGraph* representing the proof graph, *ProofNode* representing proof graph nodes, *BLinkingMonitor* and *BIntersectionMonitor* used to handle linked and intersection roles respectively.

The ProofGraph class stores the set of nodes and the set of edges corresponding to the set of nodes and the set of edges in the basic credential graph in its instance variables: *nodes* and *edges* respectively. Adding nodes and edges is handled by the *addNode()* and *addEdge()* methods. The main processing is handled by the *bProcess()* method of the ProofGraph class. The nodes to be processed are stored in the backward processing queue (*bQueue*).

Each node in the graph is represented by an instance of the ProofNode class. Each ProofNode object stores the set of backward solutions in the *bSolutions* attribute. A *solution* in the backward search algorithm is an entity. Thus, the *bSolutions* attribute of the ProofNode class stores all the entities which are known to be members of the corresponding role expression. When a new solution D is discovered, every node e such that there is a path $[e] \overset{*}{\Leftarrow} [D]$ in the proof graph must be notified about this solution. This is realized using a well-know *observer* design pattern. Every instance of the ProofNode class maintains a list of observers, called *backward solution monitors* in the text. When a node is notified about one or more new solutions – by invoking node's *notify()* operation – it immediately notifies all the monitors (observers) of the node using node's *notifyAll()* operation. Every node which is not an entity node (entities do not have any solutions other than themselves) can be *registered* as a backward monitor of a node using node's *bAttach()* operation. There are two special backward monitors that are not instances of the ProofNode class: backward linking and intersection monitors. In Listing 2.1 they are represented by two classes: *BLinkingMonitor* and *BIntersectionMonitor*. Linking and intersection monitors realise the basic credential graph closure properties 2 and 3 respectively.

When processing a linked role $A.r_1.r_2$, the algorithm first creates a new node for the role $A.r_1$, then it creates a backward linking monitor and attaches this monitor to $[A.r_1]$. The backward linking monitor works as follows: when the backward linking monitor corresponding to a linked role $A.r_1.r_2$ is notified about a new solution B, it means that B became a member of $A.r_1$. By the closure property 2 in Definition 2.3.2 this implies that the basic credential graph contains the edge $[A.r_1.r_2] \Leftarrow [B.r_2]$. The backward linking monitor realises this by creating new node corresponding to role $B.r_2$ and by adding the edge $[A.r_1.r_2] \Leftarrow [B.r_2]$ to the proof graph (lines (46–50)).

When processing an intersection node $[B_1.r_1 \cap B_2.r_2]$, the algorithm first creates two new nodes $[B_1.r_1]$ and $[B_2.r_2]$, then it creates a backward intersection monitor and attaches this monitor to these two newly created nodes. When any of these two nodes receives a new solution D, it notifies all of its backward solution monitors, including the monitor corresponding to $B_1.r_1 \cap B_2.r_2$. When this monitor is notified about solution D it checks how many times it observed the addition of entity D. When the counter reaches 2, it adds edge

$[B_1.r_1 \cap B_2.r_2] \Leftarrow [D]$ to the proof graph (lines (52–59)). *BIntersectionMonitor* has attribute *solutions* to monitor how many times the addition of a given solution was observed.

In order to find all members of a role $A.r$ the algorithm is initialised using the following sequence:

```
proofGraph = new ProofGraph(C)
proofGraph.addNode(A.r)
proofGraph.bProcess()
```

Listing 2.1: Backward Search Algorithm

```
1   class ProofGraph:
      def __init__ ():
3       clear (bQueue)
        clear (nodes)
5       clear (edges)
      def bProcess():
7       while not bQueue.empty():
          n = bQueue.dequeue()
9         if n is an entity D:
            n.bSolutions.add(D)
11          n.notifyAll(D)
            continue
13        if n is a role A.r:
            foreach A.r ⟵ e ∈ C:
15            addNode(e)
              addEdge([A.r] ⇐ [e])
17          continue
          if n is a linked role A.r1.r2:
19          n1 = addNode(A.r1)
            n1.bAttach(new BLinkingMonitor(A.r1.r2))
21          continue
          if n is an intersection B1.r1 ∩ B2.r2:
23          n1 = addNode(B1.r1)
            n2 = addNode(B2.r2)
25          m = new BIntersectionMonitor(B1.r1 ∩ B2.r2)
            n1.bAttach(m)
27          n2.bAttach(m)
            continue
29    def addNode(e):
        if nodes.contains(e): return getNode(e)
31      n = new ProofNode(e)
        nodes.add(n)
33      bQueue.enqueue(n)
        return n
35    def addEdge([e2] ⇐ [e1]):
```

```
        n1 = getNode(e1)
37      n2 = getNode(e2)
        if not edges.contains(n2 ⇐ n1):
39        edges.add(n2 ⇐ n1)
        if n1.hasSolutions():
41        s = n1.getSolutions()
          n2.bSolutions.add(s)
43        n2.notifyAll(s)
        n1.bAttach(n2)
45
    class BLinkingMonitor(A.r1.r2):
47    def notify (sols ):
        foreach B in sols :
49          proofGraph.addNode(B.r2)
            proofGraph.addEdge([A.r1.r2] ⇐ [B.r2])
51
    class BIntersectionMonitor (B1.r1 ∩ B2.r2):
53    def __init__ ():
          clear (solutions )
55    def notify (sols ):
        foreach D in sols :
57          solutions.add(D)
            if solutions.count(D) == 2:
59            proofGraph.addEdge([B1.r1 ∩ B2.r2] ⇐ [D])

61  class ProofNode:
      def __init__ ():
63      clear (bMonitors)
        clear (bSolutions )
65    def bAttach(m):
        bMonitors.add(m)
67    def notify (solutions ):
        bSolutions.add(solutions)
69      notifyAll (solutions )
      def notifyAll (solutions ):
71      foreach m in bMonitors:
          m.notify (solutions )
```

Example 2.7 Figures 2.2(a)-(d) illustrate the process of constructing the proof graph by doing backward search from *EPub.discount* for the following set of credentials $\mathcal{C}$ (a subset of Example 2.1). This corresponds to the query of Sort 1: determine the set of members of *EPub.spdiscount*, $[\![EPub.spdiscount]\!]_{SP(\mathcal{C})}$.

(1) *EPub.spdiscount* ⟵ *EOrg.preferred* ∩ *ACM.member*
(2) *EOrg.preferred* ⟵ *EOrg.university.student*
(3) *EOrg.university* ⟵ *ABU.accredited*

(4) *ABU.accredited* ⟵ *StateU*
(5) *StateU.student* ⟵ *RegistrarB.student*
(6) *RegistrarB.student* ⟵ *Alice*
(7) *ACM.member* ⟵ *Alice*

In Figs. 2.2(a)-(d), the first line of each node gives the node number (following the order of creation) and the role expression represented by the node. The second line lists the solutions associated to the node. To help the reader, we have labelled each solution and each graph edge with the number of the node that was being processed when the solution or edge was added. In each of the figures dashed edges and nodes are the newly processed nodes while the newly added solutions are grey. Below we present the process of the construction of this proof graph.

The algorithm starts the search from *EPub.spdiscount*. The only credential defining role *EPub.spdiscount* is (1). To process this credential, the algorithm adds the new node [*EOrg.preferred* ∩ *ACM.member*] to the proof graph, and inserts this node into the queue of nodes *bQueue* (lines (29–34)). Then the algorithm adds a credential edge from [*EOrg.preferred* ∩ *ACM.member*] to [*EPub.spdiscount*] (Fig. 2.2(a)). We label the edge connecting nodes [*EOrg.preferred* ∩ *ACM.member*] and [*EPub.spdiscount*] with number 0 to indicate that this edge was added while processing node [*EPub.spdiscount*]. [*EOrg.preferred* ∩ *ACM.member*] is an intersection node. To process this node, the algorithm first creates two new nodes: [*EOrg.preferred*] and [*ACM.member*], and adds them to the processing queue in this order. Next the algorithm creates an intersection monitor and attaches this monitor to both [*EOrg.preferred*] and [*ACM.member*] (lines (22–28, and the two edges labelled with 1 in Fig. 2.2(a)). This monitor guarantees that if the same solution D appears in both [*EOrg.preferred*] and [*ACM.member*], a derived edge is added from [D] to [*EOrg.preferred* ∩ *ACM.member*] (lines (52–59)). The next node to process is [*EOrg.preferred*]. The only credential defining this role is the linking inclusion *EOrg.preferred* ⟵ *EOrg.university.student*. The algorithm adds node [*EOrg.university.student*] to the graph, and a credential edge from this node to [*EOrg.preferred*] (Fig. 2.2(b)). Next, the node [*ACM.member*] is processed. Giving the presence of the credential *ACM.member* ⟵ *Alice*, the algorithm adds a new node [*Alice*] to the graph and to the processing queue. This node will be processed *after* the node [*EOrg.university.student*], so we do not add any solution at this stage.

The next node to process is [*EOrg.university.student*]. As this is a node representing a linked role, the algorithm first adds new node [*EOrg.university*] to the proof graph (and also to the processing queue) and then the algorithm attaches a linking monitor to [*EOrg.university*] (lines (18–21 and Edge 4 in Fig. 2.2(b)). This monitor behaves as follows: each time [*EOrg.university*] receives a new solution B, the monitor creates a node for *B.student* and adds the derived edge from [*B.student*] to [*EOrg.university.student*] (lines (46–50)).

The next node to process is [*Alice*]. As this is an entity node, *Alice* becomes a solution of [*Alice*] and [*Alice*] notifies all its backward solution monitors: in our case [*ACM.member*]. The intersection monitor stored by [*ACM.member*] observes that *Alice* is the received solution, but takes no action because *Alice* has been added only to [*ACM.member*], and does not appear as a solution at *EOrg.preferred* yet.

In a similar manner, [*EOrg.university*] receives the solution *StateU* when processing node [*StateU*] (Fig. 2.2(c)). After this, the linking monitor stored at [*EOrg.university*] creates the new node [*StateU.student*].

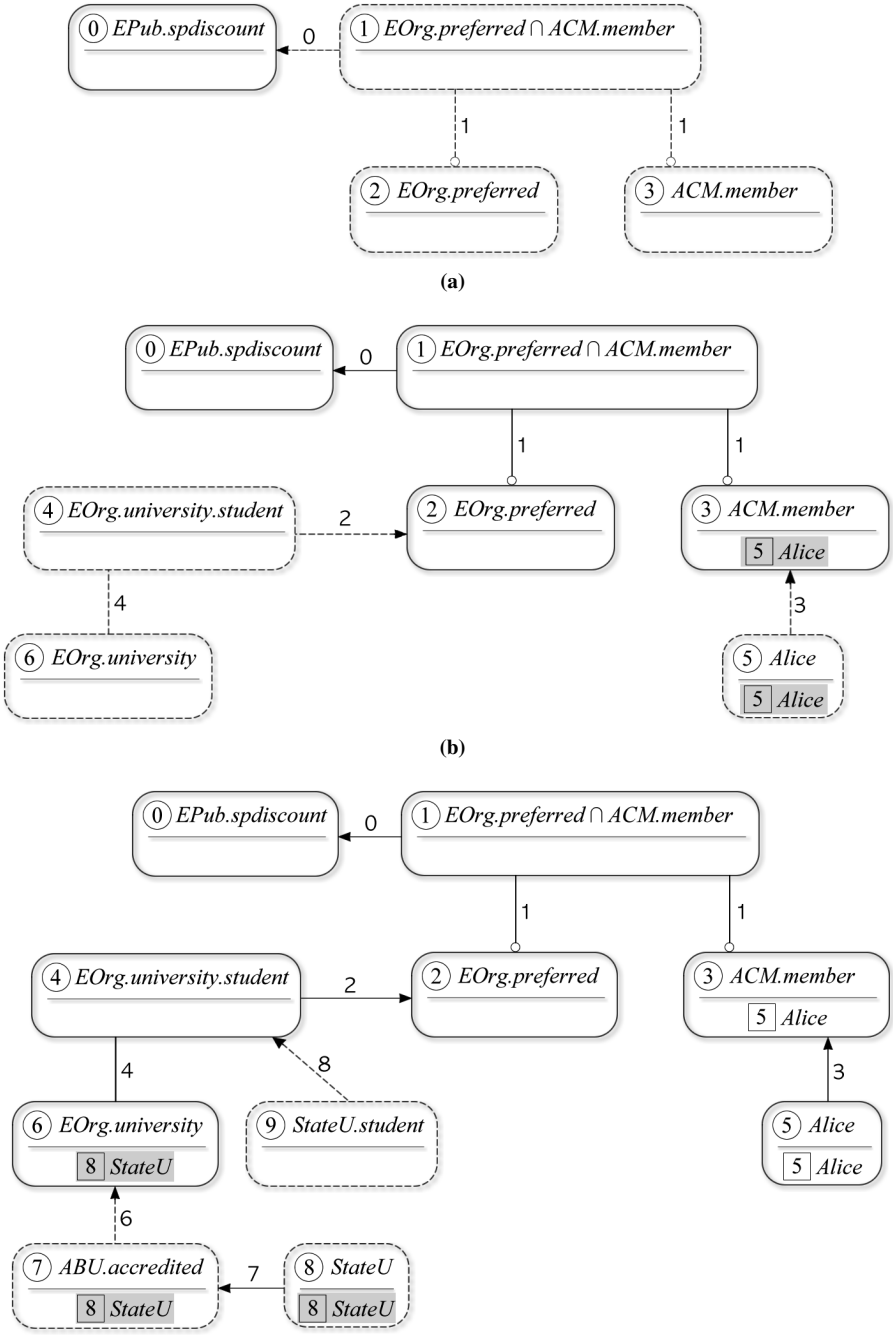
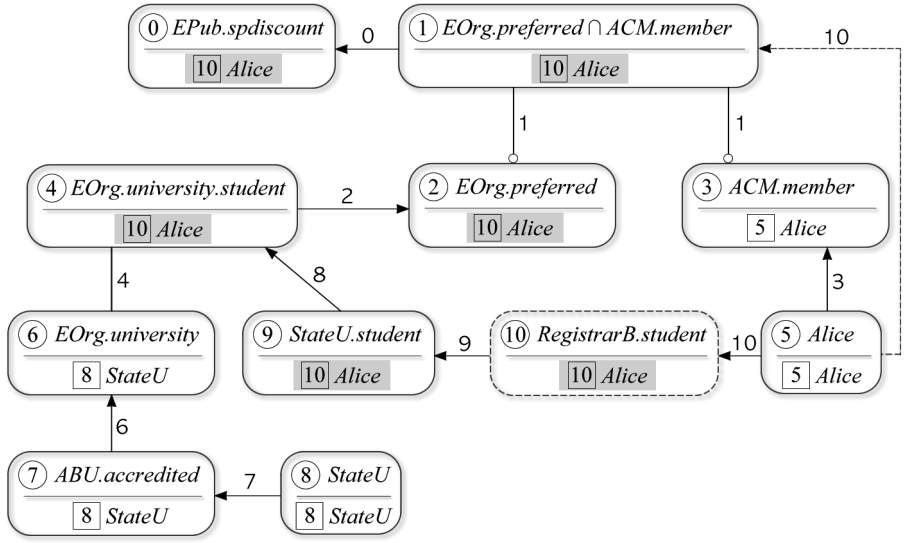


Fig. 2.2: Backward search from *EPub.spdiscount*



(d)

Fig. 2.2: Backward search from *EPub.spdiscount* cont.

When *StateU.student* receives the solution $Alice$ from [*RegistrarB.student*] (Fig. 2.2(d)), this solution is propagated upward to [*EOrg.university.student*] and [*EOrg.preferred*]. The intersection monitor at node [*EOrg.preferred*] observes that $Alice$ is added for the second time, this time by means of node [*EOrg.preferred*], and in response the monitor creates a derived edge from $[Alice]$ to [*EOrg.preferred* ∩ *ACM.member*]. The solution $Alice$ is then immediately copied from $[Alice]$ to node [*EOrg.preferred* ∩ *ACM.member*] and then to [*EPub.spdiscount*] (lines (40–43)).

At this point, there are no more nodes to process and the algorithm terminates. Given the set of credentials shown above, *EPub.spdiscount* has only one member: *Alice*.

2.3.2 The Forward Search Algorithm

The forward search algorithm answers queries of the third sort, i.e. it finds all roles that contain a given entity D_0 as a member. The direction of the search moves from the subject of a credential towards its issuer.

The forward algorithm has the same overall structure as the backward algorithm. It constructs a proof graph, maintaining a queue of nodes to be processed; both contain initially just one node, $[D_0]$. Nodes are processed one by one until the queue is empty. Listing 2.2 reports the algorithm's pseudo-code.

A solution in the forward search algorithm can be a *full solution* or a so called *partial solution*. A full solution is a role and indicates that the initial node is a member of this role. Partial solutions are necessary to properly handle intersections (see Closure Property 3 in Definition 2.3.2). Given an intersection $B_1.r_1 \cap B_2.r_2$ a partial solution has the form

$(B_1.r_1 \cap B_2.r_2, i)$ where $i \in \{1, 2\}$. We add the partial solution $(B_1.r_1 \cap B_2.r_2, i)$ to the node $[e]$ when $[B_i.r_i]$ is reachable from $[e]$ (lines (12–13)).

Similarly to the backward processing algorithm, when a node receives either a full, or a partial solution, it notifies each of its forward solution monitors. The solutions travel through the edges eventually reaching some other entity node $[D]$. When $[D]$ is notified about new partial solution $(B_1.r_1 \cap B_2.r_2, i)$, it checks whether it has the two partial solutions $(B_1.r_1 \cap B_2.r_2, 1)$ and $(B_1.r_1 \cap B_2.r_2, 2)$, and, if so, it adds a derived edge $[B_1.r_1 \cap B_2.r_2] \Leftarrow [D]$ to the proof graph (lines (50–53).

Linking roles are handled using forward linking monitors. A linking monitor is created when processing a role $B.r_2$. A new node $[B]$ is created and a forward linking monitor *FLinkingMonitor($B.r_2$)* is attached to $[B]$ (lines (16–17)). This monitor, when notified by $[B]$ about new solution $A.r_1$, creates new node $[A.r_1.r_2]$ and adds it to the proof graph and to the forward processing queue. Then, it adds new edge $[A.r_1.r_2] \Leftarrow [B.r_2]$ to the proof graph (lines (36–40)).

In order to find all roles $A.r$ an entity D_0 is a member of, the algorithm should be initialised using the following sequence:

```
proofGraph = new ProofGraph(C)
proofGraph.addNode(D)
proofGraph.fProcess ()
```

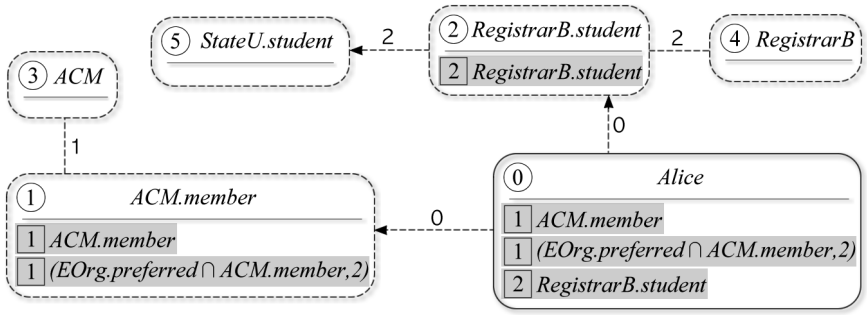
Listing 2.2: Forward Search Algorithm

```
1  class ProofGraph:
     def __init__ ():
3        clear (fQueue)
         clear (nodes)
5        clear (edges)
     def fProcess ():
7      while not fQueue.empty():
         s = ∅
9        n = fQueue.dequeue()
         if n is a role B.r2:
11         s.add(B.r2)
           foreach A.r ⟵ f1 ∩ f2 ∈ C s.t. ∃ i ∈ {1,2}, fi = B.r2:
13           s.add((f1 ∩ f2, i))
           n.fSolutions.add(s)
15         n.notifyAll(s)
           n1 = addNode(B)
17         n1.fAttach(new FLinkingMonitor(B.r2))
         # get the role expression associated with node n
19       e = n. roleExpression ()
         foreach A.r ⟵ e ∈ C:
21         addNode(A.r)
           addEdge([A.r] ⇐ [e])
23   def addNode(e):
```

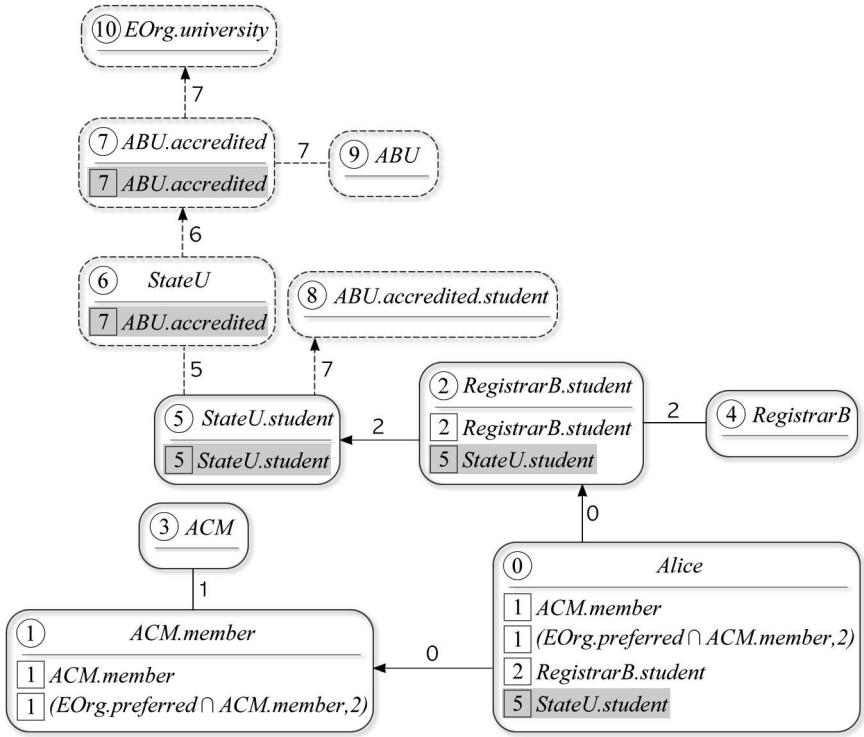
```
        if nodes.contains(e): return getNode(e)
        n = new ProofNode(e)
        nodes.add(n)
        fQueue.enqueue(n)
        return n
    def addEdge([e2] ⇐ [e1]):
        n1 = getNode(e1)
        n2 = getNode(e2)
        if not edges.contains(n2 ⇐ n1):
            edges.add(n2 ⇐ n1)
        n2.fAttach(n1)

class FLinkingMonitor(B.r2):
    def notify (solutions ):
        foreach A.r1 in solutions:
            proofGraph.addNode(A.r1.r2)
            proofGraph.addEdge([A.r1.r2] ⇐ [B.r2])

class ProofNode:
    def __init__ ():
        clear (fMonitors)
        clear (fSolutions )
    def fAttach (m):
        fMonitors.add(m)
    def notify (solutions ):
        fSolutions.add(solutions)
        if the node is an entity node D:
            foreach f1 ∩ f2 s.t. ∀i ∈ {1, 2}∃(f1 ∩ f2, i) ∈ fSolutions:
                proofGraph.addNode(f1 ∩ f2)
                proofGraph.addEdge([f1 ∩ f2] ⇐ [D])
        else : notifyAll (solutions )
    def notifyAll (solutions ):
        foreach m in fMonitors:
            m.notify (solutions )
```

Example 2.8 Figures 2.3(a)-(c) depict the process of constructing the proof graph by forward search from $[Alice]$ for the set of credentials from Example 2.1.

The first line of each node reports the node number in order of creation and the role expression represented by the node. The second part of a node lists the solutions associated to the node. Each solution and each graph edge is labelled with the number of the node that was being processed when the solution or edge was added. In each of the figures the dashed edges and nodes are the new ones and the new solutions are grayed. The process begins from node [*Alice*] (Fig. 2.3(a)). As [*Alice*] is an entity node, the algorithm searches for all credentials having *Alice* as the body. There are two such credentials: *ACM.member* $\longleftarrow$ *Alice* and *RegistrarB.student* $\longleftarrow$ *Alice*. Thus, the algorithm creates two nodes: [*ACM.member*] and [*RegistrarB.student*] and adds a credential edge from [*Alice*]

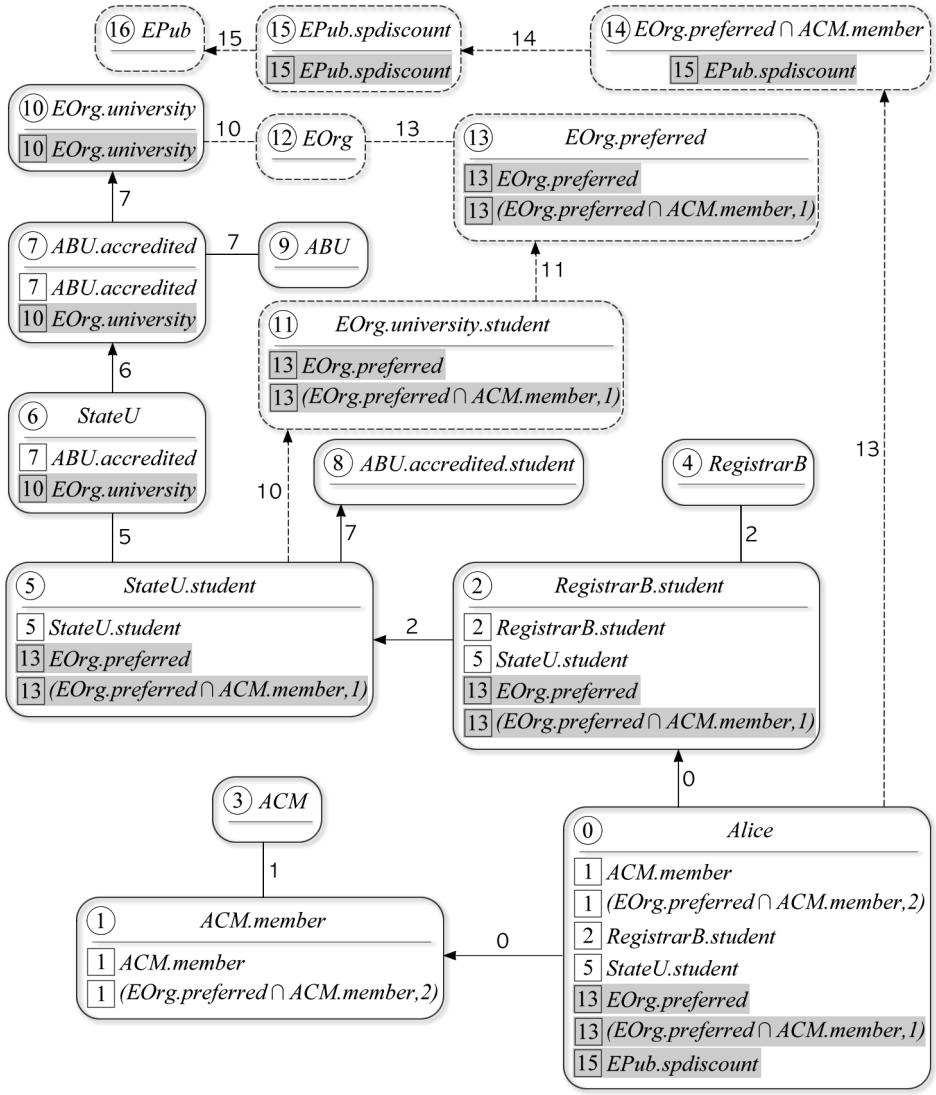


(a)



(b)

Fig. 2.3: Forward search from *Alice*



(c)

Fig. 2.3: Forward search from *Alice* cont.

to [*ACM.member*] and a credential edge from [*Alice*] to [*RegistrarB.student*]. The next node to be processed is [*ACM.member*] (recall that the number in the circle displays the order of the processing). *ACM.member* is a role. Therefore, the algorithm first adds to node [*ACM.member*] a solution *ACM.member*. Next, the algorithm checks if there are any intersection credentials having *ACM.member* in the body. The role *ACM.member* appears as the second component of *EOrg.preferred* ∩ *ACM.member* in credential (1). Thus, the algorithm adds the partial solution (*EOrg.preferred* ∩ *ACM.member*,2) to the solution space of [*ACM.member*] (lines (12–13)). The node [*ACM.member*] notifies all its forward solution monitors about the new solutions. So, [*Alice*] receives *ACM.member* and (*EOrg.preferred* ∩ *ACM.member*,2) as its first solutions. Now, the algorithm creates the node [*ACM*] and a forward linking monitor (edge with number 1 in Fig. 2.3(a)), which is then added as a solution monitor to [*ACM*] (lines (16–17)). At any time, this monitor, on observing that [*ACM*] gets a full solution $A.r$, creates the node [$A.r$*.member*] and adds the edge from [*ACM.member*] to [$A.r$*.member*] to the proof graph (lines (36–40)). The node [*RegistrarB*] is processed in a similar way. There are no credentials having *ACM* or *RegistrarB* as the body, so [*ACM*] and [*RegistrarB*] do not have any solutions. Figure 2.3(a) shows the snapshot of the graph after processing of node [*RegistrarB*].

Figure 2.3(b) shows the graph after processing of node [*ABU.accredited*]. The nodes [*ABU*] and [*EOrg.university*] are the ones added when processing [*ABU.accredited*] and are next to be processed. When [*StateU*] receives the solution [*ABU.accredited*] its (forward) linking monitor creates node [*ABU.accredited.student*].

Figure 2.3(c) shows the complete graph. [*EOrg.preferred*] has one full solution, *EOrg.preferred*, and one partial solution (*EOrg.preferred* ∩ *ACM.member*,1), which comes from the fact that *EOrg.preferred* is the first component of the intersection in the body of credential (1). [*EOrg.preferred*] notifies its forward solution monitors about these two solutions, which eventually reach [*Alice*]. When [*Alice*] is notified, since [*Alice*] has the two partial solutions corresponding to the intersection *EOrg.preferred* ∩ *ACM.member*, [*Alice*] creates the intersection node [*EOrg.preferred* ∩ *ACM.member*] and the edge from [*Alice*] to [*EOrg.preferred*∩*ACM.member*]. Finally, [*Alice*] receives the solution *EPub.spdiscount* from [*EPub.spdiscount*].

2.3.3 The Bidirectional Search Algorithm

The two algorithms presented in Sect. 2.3.1 and Sect. 2.3.2 can also be used to answer the queries of Sort 1 presented at the top of Sect. 2.3 in which given a role $A.r$ and an entity D, one wants to determine whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. This can be done either by using the backward search and starting from $A.r$ or by using forward search and staring from D. It is also possible to perform both searches at the same time. Such an algorithm is called a *bidirectional search algorithm*. This may not make too much sense at first – as the bidirectional search algorithm may construct a larger graph than does either backward or forward search – but as we show later in Sect. 2.4, this may be very useful when the credential storage is distributed.

2.4 The Storage Type System

Winsborough and Li argue that a trust management language should have support for *distributed* credential storage [101]. In our description so far, we assume that the credential storage is centralised; more precisely, we have assumed that at any time we can list the whole set of credentials. Such an assumption is not realistic in practice, as sometimes we may want to store the credentials at their issuers and sometimes at their subjects (see [67, 101] for a discussion). Intuitively, the problem with decentralised storage is that one may not know where to find the credentials needed to build a proof. Let us see an example of this.

Example 2.9 Assume that the policy contains only two credentials:

$$(1) \qquad A.r \longleftarrow B.r_1$$
$$(2) \qquad B.r_1 \longleftarrow D$$

Now, assume that one wants to know whether $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. Each of these two credentials could be stored at either its issuer and/or its subject.

First, let us assume that credential (1) is stored at A and credential (2) at D. Using backward search, we start from node $[A.r]$ by listing all credentials defining $A.r$. The only credential stored at A is $A.r \longleftarrow B.r_1$, so, the only way to proceed from here is to "go to" B, but since B does not store any credentials, the backward search algorithm concludes that $[\![A.r]\!]_{SP(\mathcal{C})}$ is empty. In the forward search algorithm we would start from $[D]$ by searching for all the credentials having D as the body. D stores only one credential:$B.r_1 \longleftarrow D$. The forward search algorithm then "goes to" B and fetches the credentials B stores. However, since B does not store any credentials, the forward search algorithm concludes that the only role D is a member of is $B.r$. Also, the forward search does not allow us to prove that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$. The bidirectional search algorithm, on the other hand, succeeds because when backward search stops at node $B.r_1$, the algorithm knows from the forward search that D is a member of $B.r_1$. Therefore, the bidirectional algorithm can conclude that D must be the member of $A.r$ as well.

Second, and perhaps more importantly, suppose that the two credentials above were stored at entity B (i.e. that (1) was stored by the subject and (2) was stored by the issuer). In this case, following the same reasoning, it is easy to see that both forward and backward search algorithms fail again, but, in addition, even the bidirectional search fails.

When both credentials are stored by their issuers (i.e. credential (1) is stored at A and credential (2) is stored at B) the only way to discover that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$ is by using backward search starting from $A.r$.

Finally, when both credentials are stored by their subjects (i.e. (1) is stored at B and (2) is stored at D) only the forward search starting from D can find out that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.

This example shows that when credential storage is distributed some chain discovery algorithms may or may not work. In particular, if credential storage is not regulated, one may be unable to find the answers to a query.

RT_0 deals with this problem by introducing a *storage type system* limiting the number of possible storage location by introducing the notion of *well-typed* credentials. Each role name r has two types: an issuer-side type and a subject-side type. On the issuer side, each role name can have one of three type values: *issuer-traces-none*, *issuer-traces-def*, and *issuer-traces-*

all. On the subject side, each role can have one of two type values: *subject-traces-none* and *subject-traces-all*. The intuition behind these type values is the following: if a role name r has the (issuer-side) type issuer-traces-all then one should be able to answer the queries of Sort 2 and to find all members of any role of the form $A.r$ using solely the backward search algorithm. Similarly, if a role name r has (subject-side) type subject-traces-all then starting from any entity D one should be able to find all roles of the form $A.r$ such that D is a member of $A.r$ (which corresponds to the queries of Sort 3). The type value issuer-traces-def is a weaker version of the issuer-traces-all type value. If a role name r has (issuer-side) type issuer-traces-def, then from any entity A one can find all credentials defining $A.r$. If a role name r has type value issuer-traces-none then for any role $A.r$, the backward search algorithm will not find any member of this role. If a role name r has type value subject-traces-none, then starting from any entity D, the forward search algorithm will not be able to find any role $A.r$ such that D is a member of $A.r$.

Summarising, we have the following definition:

Definition 2.4.1 (Type) *A* type *is a mapping from role names into two-element sets of the form* $\{i, s\}$*, such that:*

- $i \in \{$*issuer-traces-all, issuer-traces-def, issuer-traces-none*$\}$*, and*
- $s \in \{$*subject-traces-all, subject-traces-none*$\}$.

We call i the *issuer-side* type value and s the *subject-side* type value of r, denoted $itype(r)$ and $stype(r)$ respectively, and we let $type(r) = itype(r) \cup stype(r)$.

The type of a role name directly indicates the storage location of the credentials.

Definition 2.4.2 (Storage) *Let* r *be a role name and* $A.r \longleftarrow e$ *be a credential.*

- *If* $itype(r) \in \{$*issuer-traces-all, issuer-traces-def*$\}$ *then* A *must store this credential.*
- *If* $stype(r) =$ *subject-traces-all then every entity* $B \in$ *base*(e) *must store credential* $A.r \longleftarrow e$.

Notice that a credential might have to be stored both by the issuer and by the subject (this is the case e.g. when one wants to be able to answer the queries of both Sort 2 and Sort 3). The type value issuer-traces-none (resp. subject-traces-none) indicates that A (resp. any entity $B \in$ *base*(e)) does not store credential $A.r \longleftarrow e$. Notice that if a role name r is issuer-traces-none and subject-traces-none at the same time, nobody would have to store the credential $A.r \longleftarrow e$ (this is an ill-typed combination and will be ruled out in the next definition).

Let us go back to the two clauses in Example 2.9. We saw that if credential (1) was stored only by its subject and credential (2) was stored only by its issuer then any of the presented algorithms would be able to give a correct answer to the query "is D a member of $[\![A.r]\!]_{SP(\mathcal{C})}$?". In the light of Definition 2.4.2 this means that we have to avoid credentials of the form $A.r \longleftarrow B.r_1$, where $itype(r) =$ *issuer-traces-none* and $stype(r) =$ *subject-traces-none*. In order to know which combinations are "good", we have the notion of *well-typed* credentials:

Table 2.1: Well Typed RT_0 credentials

(a)

		$A.r \longleftarrow B.r_1$		
	r_1	ITA	ITD	STA
r	ITA	OK		
	ITD	OK	OK	OK
	STA			OK

(b)

		$A.r \longleftarrow A.r_1.r_2$								
	r_1	ITA			ITD			STA		
	r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	ITA	OK								
	ITD	OK	OK	OK			OK			OK
	STA									OK

(c)

		$A.r \longleftarrow B_1.r_1 \cap B_2.r_2$								
	r_1	ITA			ITD			STA		
	r_2	ITA	ITD	STA	ITA	ITD	STA	ITA	ITD	STA
r	ITA	OK	OK	OK	OK			OK		
	ITD	OK	OK	OK	OK	OK	OK	OK	OK	OK
	STA			OK			OK	OK	OK	OK

Definition 2.4.3 (Well-typed Credentials) *An RT_0 credential c is* well-typed *if no role name occurring in c has type $\{$issuer-traces-none, subject-traces-none$\}$ and:*

- *if $c = A.r \longleftarrow B.r_1$ then $\forall t \in type(r), \exists\, t_1 \in type(r_1)$ s.t. the corresponding entry in Table 2.1(a) is* OK*;*
- *if $c = A.r \longleftarrow A.r_1.r_2$ then $\forall t \in type(r), \exists\, t_1 \in type(r_1)$ and $\exists\, t_2 \in type(r_2)$ s.t. the corresponding entry in Table 2.1(b) is* OK*;*
- *if $c = A.r \longleftarrow B_1.r_1 \cap B_2.r_2$ then $\forall t \in type(r), \exists\, t_1 \in type(r_1)$ and $\exists\, t_2 \in type(r_2)$ s.t. the corresponding entry in Table 2.1(c) is* OK*.*

For example, take the credential $c : A.r \longleftarrow A.r_1.r_2$ and assume that $type(r) = type(r_1) = \{$*issuer-traces-def*, *subject-traces-all*$\}$ and that $type(r_2) =$ $\{$*issuer-traces-none*, *subject-traces-all*$\}$. Then, we see that for both type values of r, *issuer-traces-def* and *subject-traces-all*, one can find a combination of type values for r_1 and r_2 such that this combination appears as a valid type assignment in Table 2.1(b). For the issuer-side type value of r, *issuer-traces-def*, we have $itype(r_1) =$ *issuer-traces-def* and $stype(r_2) =$ *subject-traces-all*; for the subject-side type value of r, *subject-traces-all*, we have $stype(r_1) = stype(r_2) =$ *subject-traces-all*. On the other hand, if we have that $type(r) = type(r_1) = type(r_2) = \{$*issuer-traces-def*, *subject-traces-none*$\}$, then c would not be well typed as there is no valid entry for this type value assignment in Table 2.1(b). Note that simple member credentials (of the form $A.r \longleftarrow D$) are always well-typed.

The following theorem summaries the results given in [67] and shows that using well-typed credentials guarantees that the algorithms presented in Sect. 2.3 give correct answers to queries even in presence of distributed credentials.

Theorem 2.4.4 *Let $\mathcal{C}$ be a set of well typed RT_0 credentials, and r be a role name.*

- *If $itype(r) =$ issuer-traces-all then for each entity A, the backward search algorithm correctly computes $[\![A.r]\!]_{SP(\mathcal{C})}$.*
- *If $stype(r) =$ subject-traces-all then for each entity D the forward search algorithm finds all the roles $A.r$ such that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.*
- *For any given entity D, the bidirectional search algorithm can always correctly determine if $D \in [\![A.r]\!]_{SP(\mathcal{C})}$.*

Example 2.10 *Consider again the policy of Example 2.9, if $type(r) = \{$issuer-traces-all, subject-traces-none$\}$ then, according to Table 2.1, for the credential (1) to be well-typed, the type of role name r_1 must also be $\{$issuer-traces-all, subject-traces-none$\}$. By Theorem 2.4.4, one can use the backward search algorithm to compute $[\![A.r]\!]_{SP(\mathcal{C})}$. On the other hand, if $type(r) = \{$issuer-traces-none, subject-traces-all$\}$ then, for the credential (1) to be well-typed, the type of r_1 must also be $\{$issuer-traces-none, subject-traces-all$\}$. For this type assignment, Theorem 2.4.4 says that, starting from D, the forward search algorithm will discover that D is a member of $B.r_1$ and $A.r$. Finally, if $type(r) = \{$issuer-traces-def, subject-traces-none$\}$ and $type(r_1) = \{$issuer-traces-none, subject-traces-all$\}$ then one can check that $D \in [\![A.r]\!]_{SP(\mathcal{C})}$ using the bidirectional search algorithm.*

2.5 Other members of the RT family

As we have already mentioned, RT_0 is only one of the members of the RT family of TM languages. In this section we intend to give a flavour of these extensions and of the reasons why they have been introduced. We do so by presenting examples. For a full explanation of the syntax and semantics of the RT family, we refer the reader to Li et al. [66] and Li and Mitchell [65].

2.5.1 RT_1

RT_1 extends RT_0 with parameterised roles. In RT_1 a role name consists of an RT_0 role name and zero or more parameters surrounded by parenthesis. A parameter can be a constant or a variable of one of five types: integer, closed enumeration, open enumeration, float, and date and time (see Li et al. [66] for details).

Example 2.11 In CITA a project document can always be read and written by its author, no matter which policy applies to it. The remaining project members can read the document

only if approved by the document author (identifiers starting with "?" are variables).

$$CITA.accessDoc(rw,?proj,?doc) \longleftarrow CITA.owner(?doc) \cap CITA.member(?proj)$$
$$CITA.accessDoc(?access,?proj,?doc) \longleftarrow CITA.approved(?access,?doc) \cap CITA.member(?proj)$$
$$CITA.approved(?access,?doc) \longleftarrow CITA.owner(?doc).approved(?access,?doc)$$

For each data type one can create a so called *static data set*, which can be used to constrain variables in credentials. Static in this context means that the values in the value set cannot depend on credentials but must be known at the time the value set is being specified.

Example 2.12 Charles restricts the access to his picture gallery to his friends that are over 18.

$$Charles.accessPictures \longleftarrow Charles.friend(?Age{:}[18..100])$$

In the example above, the possible values of the variable *?Age* are restricted to be in the range between 18 and 100.

For the linking inclusion credentials, a parameter can also be a special keyword *this*, which refers to a potential member of a linked role.

Example 2.13 CITA gives an annual salary increase to an employee if the employee's manager says that the performance of the employee is good:

(1) $CITA.salaryIncrease \longleftarrow CITA.managerOf(this).goodPerformance$
(2) $CITA.managerOf(Marcin) \longleftarrow Sandro$
(3) $Sandro.goodPerformance \longleftarrow Marcin$

When evaluating credential (1) above, parameter *this* takes the value *Marcin*. *Marcin* is therefore a member of *CITA.salaryIncrease*.

2.5.2 RT_2

RT_2 extends RT_1 with *logical objects* that can be used to dynamically restrict possible values of the variables occurring in credentials. A logical object, or *o-set*, is similar to an RT_1 credential, but its member set is not restricted to that of entities. For instance, a company can define an o-set containing a selection of company documents, running projects, and also any other valid RT_1 entity.

Example 2.14 The policy of CITA states that any document of a project in CUS is also a document of this project in CITA.

$$CITA.document(?proj) \longleftarrow CUS.document(?proj)$$

Now, CITA allows members of a project team to read documents of this project:

$$CITA.accessDoc(read,?D{:}CITA.documents(?proj)) \longleftarrow CITA.member(?proj)$$

In the example above, *?D:CITA.documents(?proj)* shows the application of a dynamic value set. A dynamic value set is a generalisation of the static value set of which example was given in Example 2.12. Similarly to the static value set, the dynamic value set can be used to constrain variables occurring in credentials. However, when the values in a static value set are fixed, the set of values a dynamic value set contains is given by the members of an o-set used as a constraint. In the example above, the set of values the variable *?D* can take is restricted to the members of the o-set *CITA.documents(?proj)*.

2.5.3 RT$^{\mathrm{T}}$

RT$^{\mathrm{T}}$ has been introduced to support threshold and separation of duty policies. For instance, a statement "A says that an entity E is a member of $A.r$ if a member C of $B.r_1$ and a member D of $B.r_2$ says so" is an example of a threshold policy. Here C and D can be one and the same entity. In a separation of duty policy, we have a stronger condition: "A says that an entity E is a member of $A.r$ if a member C of $B.r_1$ and a member D of $B.r_2$ such that C and D are two different entities both say so". RT$^{\mathrm{T}}$ is able to express this two types of policy.

Consider the following policy presented by Li et al. [66] being a combination of a threshold and a separation of duty policy: "A says that an entity is a member of $A.r$ if one member of $A.r_1$ and two *different* members of $A.r_2$ all say so". This policy cannot be expressed in the RT dialects presented so far, and to express this in RT one needs to use the so-called *manifold roles*. Manifold roles extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A credential of the form $A.r \longleftarrow B_1.r_1 \odot B_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of $A.r$ if s_1 is a member of $B_1.r_1$ and s_2 is a member of $B_2.r_2$. Notice that both s_1 and s_2 are (possibly singleton) non-empty sets of entities (if the set contains only one element the surrounding curly brackets can be omitted). A credential $A.r \longleftarrow B_1.r_1 \otimes B_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional sorts of credentials one can express the above statement as follows:

$$A.r \longleftarrow A.r_4.r$$
$$A.r_4 \longleftarrow A.r_1 \odot A.r_3$$
$$A.r_3 \longleftarrow A.r_2 \otimes A.r_2$$

Example 2.15 In CITA, a program must be verified by two *different* testers: one from CITA and one form CUS.

$$\textit{CITA.verified} \longleftarrow \textit{CITA.testTeam.approved}$$
$$\textit{CITA.testTeam} \longleftarrow \textit{CITA.tester} \otimes \textit{CUS.tester}$$

2.5.4 RT$^{\mathrm{D}}$

The RT framework also supports the so called *delegation of role activations*, which are useful when one needs to delegate authority temporarily to a process or an agent. RT$^{\mathrm{D}}$ provides a *delegation credential* for this reason. Here we only present the basic intuition of how it works. The simplest form of delegation credential is $D \xrightarrow{D \text{ as } A.r} B_0$, which means that D delegates to B_0 the right of acting in D's behalf "as member of $A.r$". We call "D as $A.r$"

a *role activation*. In the delegation credential above, B_0 can also represent a request, rather than an entity. Consider for instance the following example:

Example 2.16 Frank is the general practitioner (GP) of Henk in the hospital of Enschede (Ziekenhuis Enschede – ZE). A general practitioner in ZE can access all medical records of his patients.

$$\textit{ZE.gp(Henk)} \longleftarrow \textit{Frank}$$
$$\textit{ZE.accessMedRec(?Patient)} \longleftarrow \textit{ZE.gp(?Patient)}$$

During his holiday in Poland, *Henk* had a serious accident and required immediate surgery in one of the hospitals in Warsaw (*WH*). *Weronika*, the operating doctor, needs to access Henk's medical records at *ZE*.

ZE and *WH* are members of the European Hospital Alliance (*EHA*). In case of necessity, a doctor from one of the associated hospitals can access the medical records of a patient of another hospital by activating her *emergency* role (this role is not active by default, and every activation is carefully logged in both the hospital and EHA logs).

$$\textit{EHA.member} \longleftarrow \textit{ZE}$$
$$\textit{EHA.member} \longleftarrow \textit{WH}$$
$$\textit{ZE.accessMedRec(?Patient)} \longleftarrow \textit{ZE.emergencyGroup.emergency(?Patient)}$$
$$\textit{ZE.emergencyGroup} \longleftarrow \textit{EHA.member}$$
$$\textit{WH.canActivateEmergency} \longleftarrow \textit{WH.doctor}$$
$$\textit{WH.doctor} \longleftarrow \textit{Weronika}$$

Weronika can activate her role *WH.emergency(Henk)* and request Henk's medical records from *ZE* using the following delegation credential:

$$\textit{Weronika} \xrightarrow{\textit{Weronika as WH.emergency(Henk)}} \textit{accessMedRec(ZE,Henk)}$$

Here notice that *accessMedRec(ZE,Henk)* is not an entity but represent an explicit request, which is then handled by a dummy entity in RT (in other words, a *dummy* process is started which issues an appropriate query to *ZE*).

2.5.5 $\text{RT}_{\ominus}$

The members of the RT family presented so far are monotonic: adding a credential to the system can only result in granting additional privileges. However, banishing negation from a TM language is not a realistic option. In fact, as stated by Li et al. [63]: "many security policies are non-monotonic, or more easily specified as non-monotonic ones". In Chapter 3 we argue that many access control decisions in complex distributed systems, like Virtual Communities (VC), are hard to model in a purely monotonic language. We propose $\text{RT}_{\ominus}$, which adds to RT a restricted form of negation called *negation in context*.

$\text{RT}_{\ominus}$ introduces a new operator $\ominus$ and the so called *exclusion* credential $A.r \longleftarrow B_1.r_1 \ominus B_2.r_2$ indicating that all members of $B.r_1$ which are *not* members of $B_2.r_2$ are members of $A.r$.

Example 2.17 Consider the policy of Example 2.5. In this policy, *Charles.friend* is defined to be a transitive closure of the set of his direct friends. Now, if for some reason Charles would like to *exclude* some entities from this set, he needs to use the following exclusion credential:

$$\textit{Charles.accessPictures} \longleftarrow \textit{Charles.friend} \ominus \textit{Charles.blackList}$$

Now, an entity is a member of Charles's *accessPicture* role if she is a member of Charles's role *friend* and she is *not* on the Charles's black list. Assume that we have:

$$\textit{Charles.blackList} \longleftarrow \textit{Sandro}$$

Then the semantics of the role *Charles.accessPictures* is:

$$[\![\textit{Charles.accessPictures}]\!]_{SP(\mathcal{C})} = \{\textit{Alice, Bob, Jeffrey, Johan}\}.$$

2.5.6 Summary

The table below summaries the key features of all the members of the RT framework.

The RT family member	Key extensions
RT_1	parameterised roles
RT_2	logical objects
RT^{T}	manifold roles and role-product operators, which can express threshold and separation of duty policies
RT^{D}	delegation of role activation, which allows for selective use of credentials
$\mathrm{RT}_{\ominus}$	restricted form of negation

RT^{D} and RT^{T} can be used, together or separately, in combination with either RT_0, RT_1, or RT_2. The resulting combinations are written $\mathrm{RT}_i^{\mathrm{D}}$, $\mathrm{RT}_i^{\mathrm{T}}$, and $\mathrm{RT}_i^{\mathrm{DT}}$ for $i = 0, 1, 2$. Currently, $\mathrm{RT}_{\ominus}$ cannot be use in connection with any other member from the table above. Notice also, that only RT_0 has a storage type system.

2.6 Related Work

In this section we briefly present other important trust management systems which influenced the design of RT framework but also present diversity of possible approaches to trust management. We also show the related work on the reputation based trust management and we show the most important differences between reputation systems and trust management.

2.6.1 Trust Management Systems

In this Section we briefly present the most important and influential trust management systems. We start with the pioneers of trust management: *PolicyMaker* [27] and *KeyNote* [25]. Then we show Simple Public Key Infrastructure/Simple Distributed Security Infrastructure

(SPKI/SDSI) [36], which is the inspiration for RT in using local name spaces. Finally, we show how the fundamental ideas of trust management are reflected in more practical systems: we refer to Cassandra [19, 20], Trust Policy Language (TPL) [54], Proof Carrying Authorisation (PCA) [8, 17, 18], and Query Certificate Manager (QCM) [53]. We also point the most important differences between the presented system and the RT framework.

PolicyMaker and KeyNote

The notion of Trust Management was introduced by Blaze et al. [27], as a problem in network security for which the authors proposed an approach based on a small collection of general principals: unified mechanism, flexibility (expressiveness), locality of control (autonomy of system participants), and separation of policy from mechanism. PolicyMaker, also designed and developed by Blaze et al. [26, 27], was the first trust management prototype system that "facilitates the development of security features in a wide range of network services." [27]

Unlike RT, PolicyMaker places very few restrictions on the specification of authorisations and delegations. Policies and credentials are fully programmable, and can be arbitrary executable programs, limited only by being strongly "sandboxed." The advantage is that the PolicyMaker approach enables application developers tremendous flexibility to define authorisations and delegations. However, its compliance checking (evaluation) is in general undecidable: no algorithm can, for each possible request, decide whether the request is authorised. There are several variants of PolicyMaker's proof of compliance problem that are proven to be decidable, but NP-hard: globally bounded proof of compliance (GBPOC), locally bounded proof of compliance (LBPOC), and monotonic proof of compliance(MPOC). A polynomial time bound can be achieved for compliance checking by combining the restrictions used in LBPOC with the requirement that assertions be monotonic. However, the constant parameters that limit computational effort expended by a legal proof of compliance are imposed arbitrarily without justification.

KeyNote [25] is a direct descendant of PolicyMaker. KeyNote assertions are written in a concise and human readable assertion language. Evaluation is based on expression evaluation, rather than on the execution of arbitrary programs, and is specified by an informal, implementation-independent semantics that defines authorisation decisions based on requested actions. Action requests are represented by a collection of variable bindings, and credentials can contain constraints on these variables that can be used to restrict the actions for which credential owners are authorised.

Credentials, in both PolicyMaker and KeyNote, bind public keys (of the credential subjects) to direct authorisations of security-critical actions. Therefore, similarly to capability-based system, KeyNote's authorisation decision procedure is simplified and does not require resolving the name or identity of the requester. However, capability-based systems are not as scalable as attribute-based systems. In capability-based systems, managing the delegation of access rights, for instance, to all students at a given university requires issuing a credential to each student for each resource to which they have access (library, cafeteria, gym, etc.). In attribute-based systems, such as RT, by utilising credentials that characterise their owners as being students, the same student ID credential can be used to authorise a wide range of actions.

SPKI/SDSI

SPKI/SDSI [36] merged the SDSI [87] and the SPKI [44] efforts together to achieve an expressive and powerful trust management system. SDSI (pronounced “sudsy"), short for “a Simple Distributed Security Infrastructure," was proposed as a new public-key infrastructure by Lampson and Rivest. Concurrently, Ellison et al. developed SPKI (pronounced “spooky"), which was an abbreviation for “Simple Public Key Infrastructure."

SDSI's main contribution is its design of linked local names, which solves the problem of determining globally unique names. In SDSI, the owner of each public key can define names local to a name space that is identified by that key. For example, “K_{Alice} friends" represents a SDSI name, where K_{Alice} is a key identifying its name space and “friends" is a name defined locally in that name space by K_{Alice}. SDSI names that start with different keys are different names, so there is only a low probability that local names in different name spaces will interfere with one another. In this way, global uniqueness of names is achieved without synchronising and coordinating naming authorities. The way in which RT's roles are defined locally, but can be referenced non-locally, is inherited from SDSI's design of local name spaces.

While SDSI is responsible for binding names to public keys, SPKI is responsible for making authorisations. SPKI's authorisation scheme can be regarded as being orthogonal to SDSI's naming scheme. Originally in SPKI, the certificate subject is represented by its public key. However, in SPKI/SDSI, the subject can be represented by its SDSI name. SDSI names provide a method to define *groups* of authorised principals, which simplifies the delegation procedure.

For example, if Bob wants to grant an authorisation to Alice's friends, Bob can simply use SDSI's group name “K_{Alice} friends". By contrast, using KeyNote, Bob would have to enumerate the public keys of every friend of Alice's in the “Licensee" field of the assertion. The flexibility obtained by using SDSI names is useful in a decentralised system. On one hand, Bob does not need to have a list of Alice's friends when he is writing the authorisation policies. On the other hand, any changes on Alice's friends list will be immediately reflected in the semantics of Bob's authorisation policies.

SPKI/SDSI's evaluator uses a bottom-up algorithm to compute a closure set containing all certificates that can be derived from the given set of certificates. A request can be authorised if it can be found in the closure set. This algorithm is proven to be polynomial [36]. However, the evaluation process must be repeated whenever any certificate has been added or revoked, or has expired, so it is not suitable for use with a large and frequently changing credential pool.

Cassandra

Cassandra [19, 20] is a role-based trust management system, which was designed with the goal of supporting the access control policies for a national electronic health record (EHR) system.

Cassandra represents policy statements by Datalog clauses with constraints. Six special predicates are predefined in Cassandra. Firstly, *canActivate*(e, r) expresses that entity e can activate role r and, as such, that e is a member of r. Secondly, *hasActivated*(e, r) indicates that entity e has activated role r. The distinction between the predicates *canActivate* and *hasActivated* corresponds to the distinction between the role membership and the session

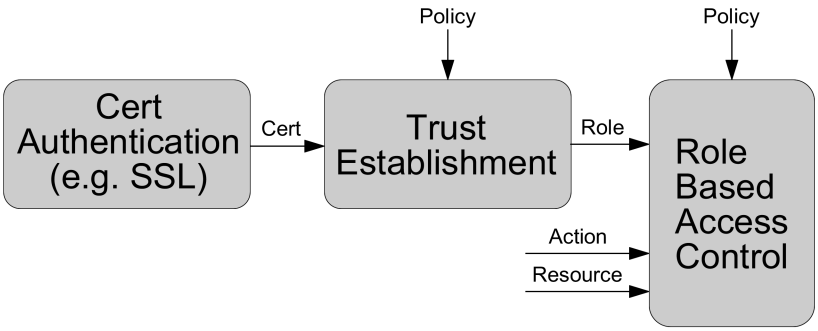


Fig. 2.4: TPL system

activation in traditional RBAC [7]. Thirdly, *canDeactivate*(e_1, e_2, r) holds if entity e_1 has the power to deactivate e_2's activation of role r. Fourthly, *isDeactivated*(e, r) becomes true if entity e's role r is deactivated. Therefore, unlike RT that can only support role membership and to some extent role activation (RT^{D}), Cassandra can also role deactivation. If a role is activated by a principal, a new fact (i.e., an atomic formula) representing this activation, and using predicate *hasActivated*, is put into the policy; similarly, deactivation of roles causes facts with predicate *hasActivated* to be removed from the policy. Fifthly, *permits*(e, a) says that the entity e is permitted to perform action a. This differs from the standard notion of role-permission assignment in two ways. On one hand, the parameter e allows constraints to refer directly to the subject of the activation. On the other hand, *permits* has no parameter for a role associated with the action, thus allowing more flexible permission specifications, e.g., a permission that is conditioned on the activation or (or perhaps merely membership in) more than one single role. Finally, *canReqCred*$(e_1, e_2.p(\overrightarrow{e}))$ says that the entity e_1 is allowed to request credentials issued by the entity e_2 and asserting the predicate $p(\overrightarrow{e})$. Besides these six special predicates, application developers can also define their own customised predicates.

TPL

TPL (Trust Policy Language) [54], designed at IBM Haifa Research Lab, was proposed specifically for trust establishment between e-strangers. TPL is based on RBAC [7] and extends RBAC by being able to map strangers to roles. Unlike RT and Cassandra, TPL's efforts are claimed to be put only into mapping users to roles, but not into mapping roles to privileges, which simplifies the design. Figure 2.4 shows the access control model proposed by Herzberg et al. [54].

In Fig. 2.4 the *Certification Authentication* module outputs the entire certificate of the requester, which becomes an input to the *Trust Establishment* (TE) system. TE maps the subject of the certificate to a role, based on the provided certificate, other certificates collected by TE and on the given role assignment policy.

TPL uses XML for application developers to write security rules, which will be translated in TPL to a standard logic programming language, e.g. Prolog. Unlike RT, which is monotonic, TPL is non-monotonic, since it includes negative rules. A negative rule

indicates that learning a new piece of knowledge (e.g., a credential) will reduce the requester's privileges. For example, a negative rule represented in Prolog statement can be "$group(X, \text{Discount}) :- \backslash+\ group(X, \text{Felon})$," in which "$\backslash+$" represents the *negation as failure*. It means that if the derivation of the credential of being a felon fails, then the requester is allowed to have the discount. However, the authors do not prove the soundness and completeness of TPL, which means that given a set of certificate one cannot guarantee that a sound decision will be made or that not every statement which is known to be true given a set of credentials will be derived by the system. The example below [54] shows the XML encoding and the corresponding Prolog translation of a TPL policy statement saying that "a hospital X can become a member of group *hospitals* if X can provide at least two *recommendation certificates* whose issuers are already known to be in the *hospitals* group and their recommendation level is higher than 1":

XML:

```
<GROUP NAME="hospitals">
    <RULE>
        <INCLUSION ID="reco" TYPE="Recommendation" FROM="hospitals"
                   REPEAT="2"/>
        <FUNCTION>
            <GT>
                <FIELD ID="reco" NAME="Level"/>
                <CONST>1</CONST>
            </GT>
        </FUNCTION>
    </RULE>
</GROUP>
```

Prolog:

$$
\begin{aligned}
&group(X, \textit{hospitals}) :- \\
&\quad cert(\textit{Y1}, X, \textit{"Recommendation"}, \textit{RecFields1}), \\
&\quad cert(\textit{Y2}, X, \textit{"Recommendation"}, \textit{RecFields2}), \\
&\quad \textit{Y1} != \textit{Y2}, \\
&\quad group(\textit{Y1}, \textit{hospitals}), \\
&\quad group(\textit{Y2}, \textit{hospitals}), \\
&\quad field(\textit{RexFields1}, \textit{"Level"}, \textit{L1}), \textit{L1} > 1, \\
&\quad field(\textit{RexFields2}, \textit{"Level"}, \textit{L2}), \textit{L2} > 1.
\end{aligned}
$$

PCA

PCA (Proof Carrying Authorization) [8, 17, 18] has been mainly designed for the access control on server's web page resources. Figure 2.5 shows the components of PCA system working in a web browsing environment. $HTTP\ proxy$ is used to make the whole process of accessing a web page transparent to the web browser. The web browser only knows the final result: either the requested web page or a denial message is displayed. The proxy is designed to be portable and easily integrated into the client system without changing anything inside the original web browser. Therefore, it saves the client from collecting relevant credentials

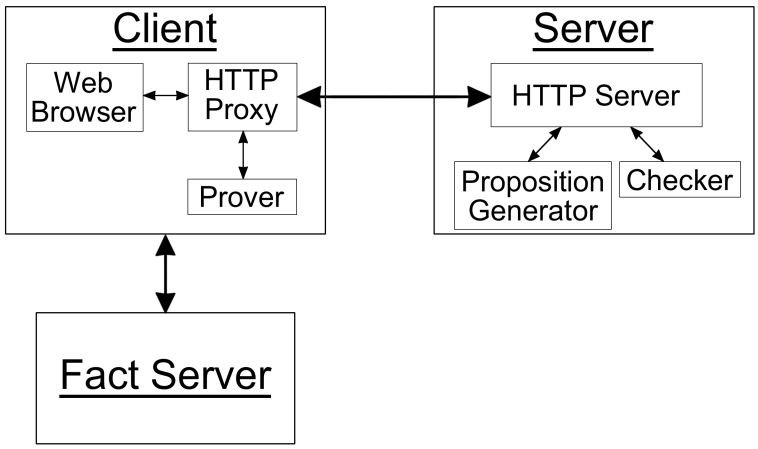


Fig. 2.5: PCA system

and negotiating with the resource owner.

PCA uses higher-order logic to specify policies and credentials, so that it can be very expressive. However, its evaluation is thus undecidable. In their design, undecidability is resolved in two phases. Firstly, in order to reduce the computation burden on the server's PCA evaluator, it is required that the requesting client constructs the authorisation proof. The server's evaluator only needs to check the proof, which is not only decidable, but can be done efficiently. Secondly, on the client side, the proxy is responsible for navigating and retrieving credentials, computing proofs and communicating with the server. In order to avoid undecidable computation at the client side, the client proxy does not use the full logic, but use an application-specific limited logic, which should be tractable.

QCM

QCM [53], short for "Query Certificate Manager", was designed at the University of Pennsylvania as a part of the SwitchWare project on active networks to support secure maintenance of distributed data sets. For example, QCM can be used to support decentralised administration of distributed repositories housing public key certificates that map names to public keys. For the purposes of access control, QCM provides security support for ACL's query and retrieval.

A QCM policy is specified in relational calculus. One of the main contributions of QCM is its design of a policy directed certificate retrieval mechanism [53], which enables the TM evaluator automatically to detect and identify missing but needed certificates, and to retrieve them from remote certificate repositories. It uses query decomposition and optimisation techniques, and its novel solutions are discussed in terms of network security, such as private key protection methods. However, unlike RT credentials, which can be stored with their either their subjects or their issuers, and can then be located and retrieved as needed during authorization evaluation, credentials in QCM must be stored with their issuers. Figure 2.6

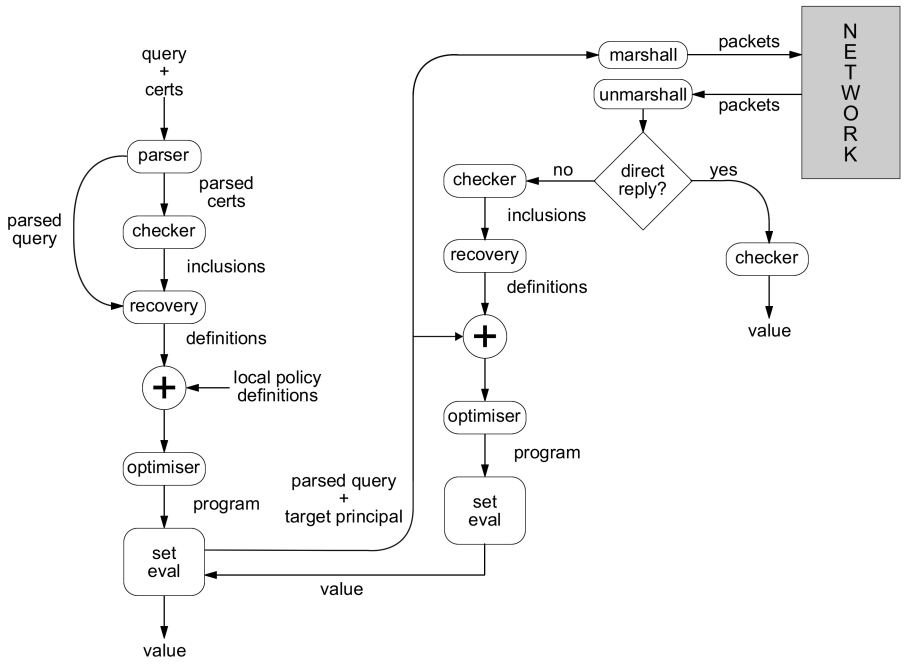


Fig. 2.6: QCM Engine

[53] shows the stages by which queries and credentials are processed in the evaluator and how the evaluation process interleaves and cooperates with the credential retrieval process.

2.6.2 Trust Management and Reputation Systems

Since the term Trust Management was first introduced by Blaze et al. [27], TM became an important and popular research area. However, in many cases the work having TM in the title has often very little in common with TM as understood by its originators. Most of these cases come from the field of *Reputation Systems* [59, 86, 92, 103, 104] – also referred in the literature as *Reputation Based Trust Management*. Although in this chapter we do not deal with reputation systems, because reputation systems are undoubtedly related to TM, we provide some background information and we mark the most evident differences with TM.

Reputation systems is now a well researched area [105]. The interest in reputation systems comes from e.g. expert and auction systems [86], like *AllExperts* (http://www.allexperts.com), where everyone can ask an expert volunteer a question from the selected area. The user can then rate the expert so that other users be informed on the quality of advice given by different experts. An example of an auction system is *eBay* (http://www.ebay.com). In eBay, every user is welcome to leave a positive, negative or neutral feedback after each transaction. Sellers and buyers in eBay can rate each others and by this they can discourage (or encourage) prospective users to enter into business with another

eBay user.

It has been observed that reputation is an important factor which naturally supports the process of building trust among people [56, 86]. The role of a reputation system is then to collect, distribute, and aggregate feedbacks (reputations) concerning participants' past behaviour [86]. The past behaviour is usually expressed using a so called *trust metric*, which describes the agent's trust in another agent - most often within some well defined context [6]. In defining trust and reputation, authors often refer to social sciences [6, 72] or economy and politics [37, 86]. In most of the formal approaches to reputation based trust management there is a clear distinction between a so called *direct* and *recommendation* trust [6, 58, 104, 105].

It is clear that the areas that both Trust Management and Reputation Systems cover overlap. There are, however, important differences. Most reputation systems are numeric [30], and do not incorporate language facilities. Reputation systems are also in general highly dynamic and deal mostly with the trust metric definition or recommendation exchange protocols. Reputation systems answer the question how to build trust values from the local history and the information provided by other peers. Most importantly, the trust gained in reputation systems is rather fuzzy in nature as it depends on an often obscure algorithm and on sometimes highly subjective feedback. In Trust Management, on the other hand, trust is obtained as a result of a formal evaluation of a set of credentials with respect to the user policy. Each user is also allowed to have different policy, which is usually not allowed in the existing reputation systems.

2.7 Conclusions

In this chapter we present the RT trust management framework. We show the syntax, the semantics and the storage type system for the core language of the RT family. We give a detailed description of the credential chain discovery algorithms supported with illustrative examples. We also present various examples demonstrating the expressive power of other members of the RT family, including the non-monotonic extension $RT_{\ominus}$ which we fully describe in Chapter 3. The contribution of this chapter, is a clearer presentation of the semantics and the type system of RT_0 than Li et al. [67], and also improved pseudo-code for the original credential chain discovery algorithms. Additionally, we believe that extensive Related Work and comparison between credential based and reputation based trust management will help the reader to understand the contents of the following chapters better.

CHAPTER 3

Nonmonotonic Trust Management for P2P Applications

In the previous chapter we presented the RT Trust Management framework showing how RT can be used to model various distributed access control scenarios. In the context of the I-Share project we needed to model virtual communities and yet we were unable to model even relatively simple scenarios involving community decisions about access control in virtual communities. The reason for this is that many access control policies in virtual communities are non-monotonic in nature. This means that they cannot be expressed in current, monotonic trust management languages of which is RT is a prominent representative.

In this chapter we propose a non-monotonic extension to RT. The new member, $RT_{\ominus}$, adds a restricted form of negation to RT, thus admitting a controlled form of non-monotonicity. In the chapter we present the declarative semantics for $RT_{\ominus}$, we extend the credential chain discovery algorithm, and provide an implementation. Finally we discuss the performance of the new algorithm when executed on the most popular Prolog engines.

The contents of this chapter was first published as M. Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. den Hartog. *Nonmonotonic Trust Management for P2P Applications*. In *Proc. 1st International Workshop on Security and Trust Management*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 101–116. Elsevier, 2005.

3.1 Introduction

Languages from the family of Role Based Trust Management Framework (RT), like most Trust Management (TM) languages are monotonic: adding a credential to the system can only result in the granting of additional privileges. Usually, this property is desirable in policy languages [89]. However, banishing negation from an access control language is not a realistic option. In fact, as stated by Li et al. [63] "many security policies are non-monotonic, or more easily specified as non-monotonic ones"; similar views are expressed by Barker and Stuckey [16] and by Wang et al. [99] in the context of logic-based access control. This is also true for complex distributed systems such as virtual communities. In particular, as we will show, modelling access control decisions by a community, as opposed to access control decisions by an individual member, cannot be made without at least a form of negation, which we call negation-in-context. As pointed out by Dung and Thang [42] a TM system should be monotonic with respect to the credential submitted by the client but could be non-monotonic with respect to the site's local information about the client. Our extension allows a TM system to be non-monotonic not only in a local setting, but also when the context for negation can be provided.

Contributions We present an enhancement to the expressive power of the RT family of trust management languages by proposing $RT_{\ominus}$, an extension of RT_0. More specifically we:

- add a single new statement type adding negation-in-context to standard RT;
- present and discuss the declarative semantics of $RT_{\ominus}$;
- show that the extension is essential to specify access control policies for virtual communities.
- describe a chain discovery algorithm for $RT_{\ominus}$.

In the next section we discuss how access control policies in virtual communities motivate us to add negation-in-context to RT. In Section 3.3 the syntax and informal semantics of $RT_{\ominus}$ is introduced. The formal semantics of $RT_{\ominus}$ is presented in Section 3.4. We present related work in Section 3.7 and conclusions and future work in Section 3.8.

3.2 Virtual Communities

Virtual communities are groups of individuals with a shared interest, relationship or fantasy [62]. The majority of current virtual communities is interested in sharing audio/video content using P2P systems [83]. Taking into account the distributed nature of virtual communities, special mechanisms for access control must be provided to ensure secure operations at both intra- and inter-community levels. As it is often impossible to identify strangers [80], trust must be established between community members and entities from outside the community prior to allowing a specific access. We adopt the solution of SPKI/SDSI [36], where cryptographic key is used as the entity identifier. This requires that each entity is the sole holder of a particular key and we rely on a Public Key Infrastructure (PKI) [33, 107] to establish the keys.

As an example imagine that Alice (A), Bob (B), and Carol (C) decide to form a virtual community (or just a community for short). At the beginning they are the only members of the community, but they welcome others to join. We represent a community by a list with an entry for each member. Each entry names the community member and the members it knows about. This knowledge results from previous interactions with the community members. In this chapter, however, when we say that one knows another community member we mean that one is capable of finding this member later if necessary. Thus, the "knows" relation is not necessarily commutative, since one entity can decide to keep track of the other, but not vice versa. For example the following list represents the community of Alice (A), Bob (B), and Carol (C):

$$A[B, C] \qquad B[A, C] \qquad C[A, B]$$

In this community all members know each other, which means that each member can locate any other member when needed. As the community grows it becomes harder and harder for each member to have complete information about all other members. Yet the community would like to protect its integrity. Rather than to require involvement of all members in decision making, a more practical and scalable approach is to allow decisions about membership to be taken by a group of coordinators selected from the community members. This group of coordinators itself forms a (sub)community. To find all the coordinators we require that the directed graph formed by the "knows" relation is strongly connected. This means that each coordinator has a relationship with *at least* one other coordinator in such a way that all coordinators can be reached. For example in the list below A knows B, B knows C and C knows B and A:

$$A[B] \qquad B[C] \qquad C[B, A]$$

To become a member of a community or to become a new coordinator *all* the existing coordinators of a given community must approve. Trust management languages based on logic programming semantics do not support queries of this kind directly. If one wants to know "if all coordinators approve entity A" without explicitly enumerating these coordinators, one must check if the *negation* of this statement - "is there any coordinator that does not approve entity A" - holds. If not, one can conclude that all coordinators approve entity A. Existing trust management languages [66] are strictly monotonic, thus do not allow for negation. For this reason they are not sufficiently expressive to model complex collaborations that commonly appear in virtual communities.

3.3 $RT_{\ominus}$

In this section we introduce a new member of the RT family: $RT_{\ominus}$. We first describe a new *role-exclusion* operator $\ominus$ and then we show how the new operator can be used to model the scenario presented in Section 3.2. The RT framework is introduced in detail in Chapter 2.

3.3.1 Extending RT_0 with negation

RT_0 and other languages from the RT framework do not support negation. As argued in Section 3.2, this limits expressiveness. Let us first see an example of negation to enforce the following separation of duty policy: "developers cannot be testers of their own code". We

would like to express in RT something similar to the LP clause:

$$\textit{verifycode}(?X) \text{ :- } \textit{tester}(?X), \textit{not developer}(?X).$$

where *?X* denotes a logical variable. This clause states that an entity A can verify the code if A is a tester and A is not the developer responsible for the code. RT^{DT} - another member of the RT framework [66] - supports thresholds and delegation of role activations; to some extent, RT^{DT} allows to model separation of concerns without using negation. However, this comes at the cost of having to define manifold roles (cumbersome to work with, in practice: recall that besides entities, a manifold role can contain also collection of entities). In any case, the examples we present in the sequel cannot be modelled in RT^{DT}. Therefore, we need a new variant of RT, which defines a new type of statement with role-exclusion operator $\ominus$:

• $A.r \longleftarrow B_1.r_1 \ominus B_2.r_2$ *(Exclusion)* All members of $B_1.r_1$ which are not members of $B_2.r_2$ are added to $A.r$.

Example 3.1 *Using the $\ominus$ operator we can solve the separation of concerns problem as follows:*

$$\textit{Company.verifycode} \longleftarrow \textit{Company.tester} \ominus \textit{Company.developer}. \tag{3.1}$$

Suppose that both Alice and Bob are testers but Alice is also a developer of the code:

$$\textit{Company.tester} \longleftarrow \textit{Alice} \quad \textit{Company.tester} \longleftarrow \textit{Bob}$$
$$\textit{Company.developer} \longleftarrow \textit{Alice}$$

We see that credential 3.1 does not make Alice a member of the Company.verifycode role. Thus, only Bob can verify the code.

3.3.2 Modelling virtual communities using $RT_{\ominus}$

Having given a simple example and its representation in $RT_{\ominus}$, we now return to the more complex scenario of community decision making from Section 3.2.

Recall that we have a community of coordinators - *Alice* (A), *Bob* (B), and *Carol* (C). Assume that another entity - say D - wants to join this community and asks *Alice* for approval. *Alice* can accept D as a new coordinator locally, but before making the final decision she must check if there is no objection from other coordinators. A coordinator expresses the objection using a so called *black list*. An entity that is on the black list of one of the coordinators will not be accepted as a new coordinator.

Table 3.1 shows the minimal definition, and the descriptions of the roles used by coordinators. We see from Table 3.1 that some roles are mandatory while the others are not. For instance the role *disagreeToAdd* must be defined by each coordinator. On the other hand, the roles *allCoord*, *allCandidates*, and *addCoord* can be defined as needed by a coordinator. Special attention must be given to the definition of the *disagreeToAdd* role. For example, a coordinator can use the following credential to say that she distrusts any entity she does not

Table 3.1: Roles used by coordinators

Definition (for coordinator A)	Description	Optional
A.agreeToAdd $\longleftarrow$ [set of entities]	A coordinator uses this role to express that she approves an entity. The role has a local meaning. It is not sufficient to be a member of the *agreeToAdd* role to become a coordinator. It is necessary that no other coordinators says that an entity is a member of her *disagreeToAdd* role. The *agreeToAdd* role, through the *allCandidates* role, provides context for the $\ominus$ operator in the definition of the the *addCoord* role.	$\times$
A.disagreeToAdd $\longleftarrow$ [see description in the text]	This role is used by a coordinator as a black list.	$\times$
A.coord $\longleftarrow$ [set of entities]	This role contains all the coordinators known by a coordinator.	$\times$
A.allCoord $\longleftarrow$ A A.allCoord $\longleftarrow$ A.allCoord.coord	This role allows a coordinator to iterate over all entities connected by the *coord* role. This role, if defined, contains all the coordinators.	✓
A.objectionToAdd $\longleftarrow$ A.allCoord.disagreeToAdd	A coordinator can use this role to obtain all entities for which there is any objection.	✓
A.allCandidates $\longleftarrow$ A.allCoord.agreeToAdd	This role, if defined, contains all the candidate coordinators locally accepted by any of the coordinators. Used as the context for the $\ominus$ operator in the body of the *addCoord* role.	✓
A.addCoord $\longleftarrow$ A.allCandidates $\ominus$ A.objectionToAdd	After becoming a member of this role, a candidate coordinator becomes a new coordinator and becomes a member of the *coord* role.	✓

accept locally:

$$A.disagreeToAdd \longleftarrow A.allCandidates \ \ominus \ A.agreeToAdd.$$

If a coordinator trusts other coordinators to select candidates she can leave the *agreeToAdd* role empty and use her *disagreeToAdd* role to block some candidates. For example, *Alice* can put E on her black list to disallow E to become a coordinator, and simultaneously accept all other candidates proposed by other coordinators:

$$A.disagreeToAdd \longleftarrow E.$$

When a candidate coordinator becomes a member of the *addCoord* role of some coordinator there must exist an additional mechanism that dynamically extends the membership of role *coord* of the same coordinator. How this mechanism is realised is implementation dependent and we discuss this further in Section 3.6.

Example 3.2 Table 3.2 shows the roles and their members as seen by Alice, Bob, and Carol. In this table, we assume that Alice agrees locally to add D as a new coordinator. Also, Bob and Carol have no objection to add D as a new coordinator, but E is on Alice's black list and F is on the black list of Bob and Carol. As a consequence, only D is the member of the *addCoord* role of Alice. Bob and Carol do not have to define the *allCoord*, *allCandidates*, *objectionToAdd*, and *addCoord* unless they themselves add a new coordinator. Notice that a coordinator must define the *allCandidates* role if she defines the *disagreeToAdd* role in terms of the *agreeToAdd* role.

Table 3.2: Adding a new coordinator - D is successful, E, F fail ("$-$" = not defined)

	coord	agreeToAdd	allCoord	allCandidates	disagreeToAdd	objectionToAdd	addCoord
Alice (A)	{B}	{D}	{A,B,C}	{D}	{E}	{E,F}	{D}
Bob (B)	{C}	{}	–	–	{F}	–	–
Carol (C)	{B,A}	{}	–	–	{F}	–	–

3.4 Semantics

The semantics of trust management languages is typically given by a translation into a logic programming (LP) language [66]. We will follow the same route. Trust management credentials are by definition distributed among different principals. The use of negation creates an additional difficulty, also because in logic programming various different semantics exist to cope with negation. We have chosen to use the Well-Founded (WF) semantics [49] for the reasons given below.

The first reason is syntactic: in a TM system it is impossible to avoid circular references, and we cannot expect policies to be (locally) *stratified*. Stratification basically means that one can restructure a logic program into separate parts in such a way that negative references from one part refer only to previously defined parts. Without the possibility of local stratification we cannot use the *perfect model semantics* [84]. For the same reason, we certainly have to use a *three valued semantics*: next to the truth values *true* and *false*, we have to admit the valued *undefined*. In short, this is because we cannot expect the completion of a policy to be a consistent logic program in the sense described in [91].

A second point is handling of positive circular references, as in $\{A.r \longleftarrow B \quad A.r \longleftarrow B.r \quad B.r \longleftarrow A.r\}$. In this case, in accordance with the semantics of RT_0, we want to be able to say that for some C, it does *not* belong to $A.r$. This forces us to exclude Kunen's semantics [61] (i.e. the semantics of logical consequences of the completion of the program together with the weak domain closure assumptions), and Fitting's semantics [48]: in both

semantics the query "does C belong to $A.r$?" would return *undefined*. The WF semantics does return false for this membership query.

Finally, the WF semantics imposes no restrictions on the syntax of programs, provides an *unique* model for each program (as opposed to e.g. the stable model semantics [50]) and enjoys an elegant fixed-point construction.

3.4.1 Well-founded Semantics

We now summarise the WF semantics [49] for general logic programs. A general logic program is defined as follows:

Definition 3.4.1 *A general logic program is a finite set of clauses of the form:* A :- $L_1, \ldots, L_n$.

Here A is an atom (the *head* of the clause) and $L_1, \ldots, L_n$ with $n \geq 0$ are literals forming the *body*. A literal is an atom A (positive literal) or a negated atom $\neg A$ (negative literal). We refer to general logic programs (GLP) simply as programs. A *fact* is a clause with an empty body. The *Herbrand universe* of a program P, denoted by $\mathcal{U}_P$, is the set of all ground terms (i.e. variable-free) constructed from constants and function symbols in P. The *Herbrand base* of P, denoted by $\mathcal{B}_P$, is the set of ground atoms obtained from predicates in P and terms in $\mathcal{U}_P$. An *instantiated clause* of P is a ground clause obtained by substituting terms in $\mathcal{U}_P$ for variables in the clause in P. The *Herbrand instantiation* $Ground(P)$ of P is the set of all instantiated clauses. A set I of ground literals is *consistent* if there is no atom A such that both A and $\neg A$ are in I. An *interpretation* of P is a consistent set of ground literals in $\mathcal{B}_P$. Intuitively, atom A is true in I if $A \in I$, false in I if $\neg A \in I$, and undefined in I if neither $A \in I$ nor $\neg A \in I$.

The well-founded semantics uses *unfounded sets* to derive atoms that are false:

Definition 3.4.2 *Let P be a program, I be an interpretation of P, and U be a subset of $\mathcal{B}_P$. U is an unfounded set of P with respect to I if every atom $A \in U$ satisfies the following condition: for every instantiated clause A :- $L_1, \ldots, L_n \in Ground(P)$ whose head is A, either (1) some literal L is false in I or (2) some positive literal L is also in U.*

Intuitively, an unfounded set is a set of atoms which can be simultaneously declared false without having to assume anything to be true. The union of unfounded sets is an unfounded set, and the greatest unfounded set of P with respect to I, denoted by $\mathbf{U}_P(I)$, is the union of all unfounded sets of P with respect to I.

Definition 3.4.3 *Let P be a program, I be an interpretation of P. Transformations $\mathbf{T}_P$ and $\mathbf{W}_P$, and the well-founded semantics of P are defined as follows:*

- *$A \in \mathbf{T}_P(I)$ if and only if there is A :- $L_1, \ldots, L_n \in Ground(P)$ such that all literals $L_1, \ldots, L_n$ are true in I;*
- *$\mathbf{W}_P(I)$ is a union of $\mathbf{T}_P(I)$ and $\neg\mathbf{U}_P(I)$ which contains the negation of each element in $\mathbf{U}_P(I)$: $\mathbf{W}_P(I) = \mathbf{T}_P(I) \cup \neg\mathbf{U}_P(I)$.*
- *The well-founded model of P is the least fixed point of $\mathbf{W}_P(I)$.*

Transformations $\mathbf{T}_P(I)$, $\mathbf{U}_P(I)$, and $\mathbf{W}_P(I)$ are monotonic [49]. Let α range over all countable ordinals, the set I_α, whose elements are literals in the Herbrand base of P, is defined recursively by: $I_{\alpha+1} = \mathbf{W}_P(I_\alpha)$. I_α is also a monotonic sequence. $I_\infty = \cup_\alpha I_\alpha$ is the least fixed point of $\mathbf{W}_P$.

Example 3.3 *Consider the program $\mathcal{P}$ with the following clauses:*

$$p \text{ :- } q. \qquad q \text{ :- } p. \qquad r \text{ :- } \neg q. \qquad s \text{ :- } \neg t. \qquad t \text{ :- } \neg s. \qquad u \text{ :- } \neg s.$$

In the well-founded model of $\mathcal{P}$ we have that p and q are false, r is true, and, because s and t are defined in terms of their own each other's complement, s and t are undefined. Then u is undefined because s is undefined. (On the other hand, all predicates would be undefined in Kunen's semantics.)

3.4.2 Translating $RT_{\ominus}$ to GLP

We first give the translation to LP for RT_0 and, using this translation, the semantics of a set of RT_0 policy statements. Next we extend this to a translation from $RT_{\ominus}$ to GLP and the semantics for a set of $RT_{\ominus}$ policy statements.
The semantics of a set of RT_0 policy statements is commonly defined by translating it into a logic program [66]. Here, we depart from the approach of Li et al. [66] by referring to the role names as predicate symbols. This removes the need for "polymorphic" mode system in Core TuLiP (Core TuLiP is introduced in Chapter 4) and also removes the temptation of (accidently) using a variable symbol for the role name. For example, the statement $A.r \longleftarrow D$ is translated to $r(A, D)$ in the Prolog program. Intuitively, $r(A, D)$ means that D is a member of the role $A.r$.

Definition 3.4.4 *Given a set $\mathcal{P}$ of RT_0 policy statements, the* semantic program*, $SP(\mathcal{P})$, for $\mathcal{P}$ is the logic program defined as follows (recall that symbols starting with "?" represent logical variables):*

- *For each $A.r \longleftarrow D \in \mathcal{P}$ add to $SP(\mathcal{P})$ the clause $r(A, D)$*
- *For each $A.r \longleftarrow B.r_1 \in \mathcal{P}$ add to $SP(\mathcal{P})$ the clause $r(A, ?Z)$:- $r_1(B, ?Z)$*
- *For each $A.r \longleftarrow A.r_1.r_2 \in \mathcal{P}$ add to $SP(\mathcal{P})$ the clause $r(A, ?Z)$:- $r_1(A, ?Y)$, $r_2(?Y, ?Z)$*
- *For each $A.r \longleftarrow B_1.r_1 \cap B_2.r_2 \in \mathcal{P}$ add to $SP(\mathcal{P})$ the clause $r(A, ?Z)$:- $r_1(B_1, ?Z)$, $r_2(B_2, ?Z)$*

The semantics *of a role $A.r$ is a set of members Z that make the predicate $r(A, Z)$ true in the semantic program: $[\![A.r]\!]_{\mathcal{P}} = \{Z \mid SP(\mathcal{P}) \models r(A, Z)\}$.*

We write $SP(P) \models r(A, Z)$ if $r(A, Z)$ is true in the unique well-founded model of P. (For negation-free programs this model coincides with the least Herbrand model used for the semantics of RT_0 by Li at al [66].) We now extend the translation of RT_0 to that of $RT_{\ominus}$ by adding the translation of the exclusion rule.

Definition 3.4.5 *Given a set $\mathcal{P}$ of $RT_{\ominus}$ policy statements, the* semantic program*, $SP(\mathcal{P})$, for $\mathcal{P}$ is the* general *logic program defined as follows:*

- *For each $A.r \longleftarrow B.r_1 \ominus B.r_2 \in \mathcal{P}$ add to $SP(\mathcal{P})$ the clause $r(A, ?Z)$:- $r_1(B_1, ?Z)$, $\neg r_2(B_2, ?Z)$*

- *All other rules are as in definition 3.4.4.*

The semantics *of a role $A.r$ is a set of members Z that make the predicate $r(A, Z)$ true in the semantic program:* $[\![A.r]\!]_{\mathcal{P}} = \{Z \mid SP(\mathcal{P}) \models r(A, Z)\}$

Note that, unlike before, the value of the semantical program may give value 'undefined' for $r(A, Z)$. In this case the agent Z is not considered to be a member of the role, nor of the negated role.

Example 3.4 *Consider a system with entities A, B, C, D, roles $A.r, B.r$ and $C.r$ and the following policy rules:*

$$A.r \longleftarrow B.r \ominus C.r \qquad C.r \longleftarrow B.r \ominus A.r \qquad B.r \longleftarrow D$$

Here D is a member of $B.r$, however, D is not a member of either $A.r$ or $C.r$. Note that as a result we have that despite the presence of the rule $A.r \longleftarrow B.r \ominus C.r$ the role $B.r$ can have members that are neither in $A.r$ nor in $C.r$.

The rules for $A.r$ and $C.r$ in the example above are referred to as negative circular dependencies; $A.r$ depends negatively on $C.r$ and $C.r$, in turn, depends negatively on $A.r$. The example shows that care is required when reasoning about policies which have negative circular dependencies.

3.4.3 Virtual Communities - translation to GLP

Having introduced an example of virtual community decision making in Section 3.2, its formalism in Subsection 3.3.2, we now give the GLP semantics of the example. Translating $RT_{\ominus}$ credentials to GLP is straightforward using the rules presented in Subsection 3.4.2.

Table 3.3 presents a complete policy and the corresponding GLP rules. If one asks Alice to add D to the group of coordinators she needs to check if D is a member of the *A.addCoord*. This is equivalent to checking whether *addCoord(A,D)* holds after the translation to GLP. She does this by checking whether D is a logical consequence of the semantic program $SP(\mathcal{P})$ by first finding the semantics of the role *A.addCoord* and checking if it contains entity D. The semantics of the role *A.addCoord* with respect to the program $\mathcal{P}$ is as follows:

$$[\![A.addCoord]\!]_{\mathcal{P}} = \{D\}.$$

The semantics of the roles *A.allCandidates* and *A.objectionToAdd* (these roles define the role *A.addCoord*) are shown below:

$$[\![A.allCandidates]\!]_{\mathcal{P}} = \{D\} \quad [\![A.objectionToAdd]\!]_{\mathcal{P}} = \{E, F\}.$$

The semantics of a role may also be an empty set: $[\![B.agreeToAdd]\!]_{\mathcal{P}} = \{\}$.

Table 3.3: Virtual Community - translation to GLP

$RT_{\ominus}$ rules	GLP semantics
A.addCoord $\longleftarrow$ A.allCandidates $\ominus$ A.objectionToAdd	addCoord$(A, ?Y)$:- allCandidates$(A, ?Y)$, $\neg$objectionToAdd$(A, ?Y)$.
A.allCandidates $\longleftarrow$ A.allCoord.agreeToAdd	allCandidates$(A, ?Y)$:- allCoord$(A, ?Z)$, agreeToAdd$(?Z, ?Y)$.
A.objectionToAdd $\longleftarrow$ A.allCoord.disagreeToAdd	objectionToAdd$(A, ?Y)$:- allCoord$(A, ?Z)$, disagreeToAdd$(?Z, ?Y)$.
A.disagreeToAdd $\longleftarrow$ A.allCandidates $\ominus$ A.agreeToAdd	disagreeToAdd$(A, ?Y)$:- allCandidates$(A, ?Y)$, $\neg$agreeToAdd$(A, ?Y)$.
A.allCoord $\longleftarrow$ A.allCoord.coord	allCoord$(A, ?Y)$:- allCoord$(A, ?Z)$, coord$(?Z, ?Y)$.
A.allCoord $\longleftarrow$ A	allCoord(A, A).
A.coord $\longleftarrow$ B	coord(A, B).
B.coord $\longleftarrow$ C	coord(B, C).
C.coord $\longleftarrow$ B	coord(C, B).
C.coord $\longleftarrow$ A	coord(C, A).
A.agreeToAdd $\longleftarrow$ D	agreeToAdd(A, D).
A.disagreeToAdd $\longleftarrow$ E	disagreeToAdd(A, E).
B.disagreeToAdd $\longleftarrow$ F	disagreeToAdd(B, F).
C.disagreeToAdd $\longleftarrow$ F	disagreeToAdd(C, F).

3.5 Credential Chain Discovery

In this section we extend the standard chain discovery algorithm to $RT_{\ominus}$ following the construction of the well-founded semantics. Recall that the definition of a role $A.r$ is the set of all credentials with head $A.r$. We assume that A stores (or at least, is able to find) the complete definition of each of her roles $A.r$, i.e. that the credentials involved are issuer-traceable. The main difficulty in the chain discovery is to obtain that B is not a member of a linked role $A.r.r'$. For this we need to check that every potential member C of $A.r$ does not have B in its role $C.r'$. So who are the potential members of $A.r$? Thanks to negation in context we can provide a reasonable overestimation of this set using chain discovery for RT_0:

Definition 3.5.1 *For a policy $\mathcal{P}$ the context policy $\mathcal{P}+$ is the policy obtained by replacing each credential of the form $A.r \longleftarrow B_1.r_1 \ominus B_2.r_2 \in \mathcal{P}$ by $A.r \longleftarrow B_1.r_1$ and leaving the other credentials unchanged. We call $[\![A.r]\!]_{\mathcal{P}+}$ the* context *of the role $A.r$.*

In the algorithm below we build a set of credentials $\mathcal{C}$ together with a set of context membership facts $\mathcal{I}+$ and a set of positive and negative membership facts $\mathcal{I}$.
Step 1. Initialise $\mathcal{I} = \emptyset$, $\mathcal{I}+ = \emptyset$ and $\mathcal{C}$ = the definition of role $A.r$.

Step 2. Discover context and credentials (classical chain discovery for $\mathcal{I}+$ and $\mathcal{C}$).
We look for new credentials top down; any credential that could possibly be relevant for role $A.r$ is added to $\mathcal{C}$. We look for the context of $A.r$ bottom up; any fact that can be derived from the credentials that we have found is added to $\mathcal{I}+$. Repeat the following until no changes occur: For each credential of the following form in $\mathcal{C}$:

$[B.r_0 \longleftarrow C]$ add $r_0(B, C)$ to $\mathcal{I}+$

$[B.r_0 \longleftarrow C.r_1]$ add the definition of $C.r_1$ to $\mathcal{C}$ and add $r_0(B, D)$ to $\mathcal{I}+$ for all $r_1(C, D)$ in $\mathcal{I}+$

$[B.r_0 \longleftarrow C_1.r_1 \cap C_2.r_2]$ add the definitions of $C_1.r_1$ and $C_2.r_2$ to $\mathcal{C}$ add $r_0(B, D)$ to $\mathcal{I}+$ whenever $r_1(C_1, D)$ and $r_2(C_2, D)$ in $\mathcal{I}+$.

$[B.r_0 \longleftarrow C.r_1.r_2]$ add the definition of $C.r_1$ and, for each $r_1(C, D) \in \mathcal{I}+$, the definition of $D.r_2$ to $\mathcal{C}$. Add $r_0(B, D)$ to $\mathcal{I}+$ whenever for some Y we have $r_1(C, Y)$ and $r_2(Y, D)$ in $\mathcal{I}+$.

$[B.r_0 \longleftarrow C_1.r_1 \ominus C_2.r_2]$ add the definitions of $C_1.r_1$ and $C_2.r_2$ to $\mathcal{C}$, add $r_0(B, D)$ to $\mathcal{I}+$ for every $r_1(C_1, D)$

Step 3. Discover positive facts in $\mathcal{I}$ (extended chain discovery 1).
We update $\mathcal{I}$ similar to $\mathcal{I}+$ in the previous step, only the last case ($\ominus$) changes. Repeat until $\mathcal{I}$ does not change, for credentials in $\mathcal{C}$ of the following form:

$[B.r_0 \longleftarrow C]$ add $r_0(B, C)$ to $\mathcal{I}$

$[B.r_0 \longleftarrow C.r_1]$ add $r_0(B, D)$ to $\mathcal{I}$ for all $r_1(C, D)$ in $\mathcal{I}$

$[B.r_0 \longleftarrow C_1.r_1 \cap C_2.r_2]$ add $r_0(B, D)$ to $\mathcal{I}$ whenever $r_1(C_1, D)$ and $r_2(C_2, D)$ in $\mathcal{I}$.

$[B.r_0 \longleftarrow C.r_1.r_2]$ Add $r_0(B, D)$ to $\mathcal{I}$ whenever for some Y we have $r_1(C, Y)$ and $r_2(Y, D)$ in $\mathcal{I}$.

$[B.r_0 \longleftarrow C_1.r_1 \ominus C_2.r_2]$ add $r_0(B, D)$ to $\mathcal{I}$ whenever $r_1(C_1, D) \in \mathcal{I}$ and either $(\neg r_2(C_2, D)) \in \mathcal{I}$ or $r_2(C_2, D) \notin \mathcal{I}+$.

Step 4. Discover negative facts in $\mathcal{I}$ (extended chain discovery 2).
We search for facts which are useful when negated in $\mathcal{I}$: Initialise $\mathcal{U} = \emptyset$. We say an atom $r(X, Y)$ is *not yet false (NYF)* if it is a member of the context and not assumed or known to be false, i.e. $r(X, Y) \in \mathcal{I}+$, $r(X, Y) \notin \mathcal{U}$ and $\neg r(X, Y) \notin \mathcal{I}$. A fact $r_2(C_2, D)$ is useful if it is not yet false and $\neg r_2(C_2, D)$ can be used to derive a fact, i.e. $B.r_0 \longleftarrow C_1.r_1 \ominus C_2.r_2 \in \mathcal{C}$ and $r_1(C_1, D) \in \mathcal{I}$. Choose one useful fact and add it to $\mathcal{U}$.

Step 4a. Following the well-founded semantics, we now show that facts in $\mathcal{U}$ are false by showing that no rule can possibly derive a fact in $\mathcal{U}$. To achieve this we may need to assume that other facts are also false, i.e. add them to $\mathcal{U}$.
For each fact $r(B, D)$ in $\mathcal{U}$ and matching rule $B.r \longleftarrow e \in \mathcal{C}$ perform:

$[B.r \longleftarrow C]$ Do nothing.

$[B.r \longleftarrow C.r_1]$ This rule cannot be used to derive $r(B, D)$ if $r_1(C, D)$ is false thus if $r_1(C, D)$ is NYF then add it to $\mathcal{U}$.

$[B.r \longleftarrow C_1.r_1 \cap C_2.r_2]$ If $r_1(C_1, D)$ and $r_2(C_2, D)$ are both NYF then choose one to add to $\mathcal{U}$.

$[B.r \longleftarrow C_1.r_1 \ominus C_2.r_2]$ If $r_1(C_1, D)$ is NYF and $r_2(C_2, D) \notin \mathcal{I}$ then add $r_1(C_1, D)$ to $\mathcal{U}$.

$[B.r \longleftarrow C.r_1.r_2]$ For all Y with $r_1(C, Y)$ NYF: If $r_2(Y, D)$ is NYF choose one of $r_1(C, Y)$ and $r_2(Y, D)$ and add it to $\mathcal{U}$.

Try each possible choice in the Step 4a above and if the resulting $\mathcal{U}$ has no elements in common with $\mathcal{I}$ then add $\neg\mathcal{U}$ to $\mathcal{I}$.

Repeat steps 3 and 4 until $\mathcal{I}$ remains unchanged.
(End of algorithm.)

The algorithm correctly finds the members of the role $A.r$ as stated by the next theorem.

Theorem 3.5.2 (Soundness and completeness) *The output $\mathcal{I}$ of the algorithm satisfies:*
$\forall B : r(A, B) \in \mathcal{I} \iff B \in [\![A.r]\!]_{\mathcal{P}}$.
where $\mathcal{P}$ is the policy containing all credentials and $A.r$ is the role being discovered.

Proof. [Sketch] The algorithm follows exactly the construction of the well-founded semantics except that only part of the interpretation is found. Thus clearly $\mathcal{I}$ will be a subset of the well-founded model for $SP(\mathcal{P})$ (giving soundness). However, as the part of the interpretion used basically covers the contexts of $A.r$ and all roles used to define $A.r$ it also covers any membership facts for $A.r$ (giving completeness).

3.6 Implementation

In the current prototype storage is centralised and we assume that all credentials can be traced by the issuer. It means that we can use slightly modified backward search algorithm presented in Chapter 2 for the discovery of the contex in Step 2 of the algorithm presented in Sect. 3.5. In such a case, Linear resolution with Selection function for General logic programs (SLG) resolution of XSB prolog can be used to compute answers to queries according to the WF model for $RT_{\ominus}$ [34]. XSB is a research-oriented, commercial-grade Logic Programming system for Unix and Windows-based platforms. XSB provides standard prolog functionality but also supports negations and constraints. Using SLG resolution XSB prolog can correctly answer queries for which standard prolog gets lost in an infinite branch of a search tree, where it may loop infinitely. A number of interfaces to other software systems including Java and ODBC are available. DLV datalog [43] and the Smodels system [75] can also be used to provide an initial implementation of $RT_{\ominus}$. The DLV system [43] is a system for disjunctive logic programs. It is distributed as a command line tool for both Windows and Linux operation systems. DLV is capable of dealing with disjunctive logic programs without function symbols allowing for strong negations, constraints and queries. DLV uses two different notions of negation: negation as failure and true (or explicit) negation. By default, DLV handles negation as failure by constructing the stable model semantics for the program. This standard behaviour can be changed using a command line option and then a WF model is built instead. The true or explicit negation expresses the facts that explicitly are known to be false. On the contrary, negation as failure does not support explicit assertion of falsity. Models of programs containing true negation are also called "answer sets". The Smodels system [75] provides an implementation of the well-founded and stable model semantics for range-restricted function-free normal programs. The Smodels system allows for efficient handling of non-stratified ground programs and supports extensions including built-in functions, cardinality, and weight constraints. The Smodels system is available either as a C++ library that can be called from user programs or as a stand-alone program with default front-end (*lparse*).

Dynamic Behaviour Apart from providing an implementation of the credential chain discovery algorithm one should also consider how and when an entity becomes the member

of the *coord* role of a given coordinator.

In our scenario, each entity can add a new coordinator to the set of coordinators. One possible sequence of operations for a coordinator A to introduce a new coordinator D could be the following:

1. A adds D to A.*agreeToAdd*.
2. A computes the member set of role A.*addCoord* (and by this also the member set of roles A.*allCandidates*, A.*objectionToAdd*, and A.*allCoord* which also are volatile roles).
3. A checks if D is a member of A.*addCoord*.
4. If D is a member of A.*addCoord* it means that there is no objection at the moment for D to become a coordinator. In this case, A adds D to A.*coord* and removes D from A.*agreeToAdd*.
5. If D is not a member of A.*addCoord* it means that at least one coordinator put D on the black list (i.e. has D as a member of its *disagreeToAdd* role). In such a case A removes D from A.*agreeToAdd*.

In theory, we prefer that each role is monotonic: once an entity becomes a member of a role it stays there forever. In practice we cannot expect to be able to forbid someone to remove a member from a role. Even in a system like ours, which supports only a restricted form of monotonicity, some roles are inherently dynamic. In our example scenario for instance, an entity D might be a member of role A.*addCoord* at some moment, but will not be five seconds later if, in the meantime, some other coordinator added D to its *objectionToAdd* role. Clearly, depending on the actual application, the entities must agree on the way the roles are used: what is sufficient in one application may be intolerable in another.

3.7 Related Work

So far little attention has been given to trust management in virtual communities. Most of the existing approaches focus on reputation-based trust models in P2P networks [92]. Abdul-Rahman and Hailes [6] propose a trust model that is based on real world social trust characteristics. They also find formal logic based trust management to be ill suited as a general model of trust. To prove this claim Abdul-Rahman and Hailes refer to the early work of Burrows and Abadi [32], and Gong, Needham, and Yahalom [52], which are more relevant to formal protocol verification than to formal reasoning on trust management. To support their work Abdul-Rahman and Hailes claim that logic based trust management systems are not suitable to be automated - the existing literature on automated trust negotiation (ATN) yields a contradictory statement (see Seamons et al. [89]). Pearlman et al. [80] present a Community Authorisation Service - a central management unit for a community that helps to enforce the policy of a virtual community. Such a central point of responsibility does not fit well in the spirit of P2P networks because of their highly distributed nature. Pearlman et al. also require that there a centralised policy exists for a virtual community. However, the policy of a virtual community may have a distributed character and can be seen as a product of the policies of the community members. Boella and van der Torre [29] take the same direction

and emphasise the distinction between authorisations given by the Community Authorisation Service and permissions granted by resource providers in virtual communities of agents. They regard authorisation as a means used by community authorities to regulate the access of customers to resources that are not under control of these authorities. According to Boella and van der Torre, permission can be granted only by the actual resource owner.

As we conclude in Section 3.2, virtual communities are also not supported by the existing trust management languages, even though the general requirements for such languages have been investigated [89].

Herzberg et al. propose in [54] a prolog-based trust management language (DTPL) together with a non-monotonic version of it (TPL). Their approach is different from ours in the sense that TLP allows for *negative certificates* namely "certificates which are interpreted as suggestions not to trust a user". This far-reaching approach leads to a more complex logical interpretation, which includes conflict resolution. As opposed to this, our approach is technically simpler and enjoys a well-established semantics. Jajodia et al. [55], Wang et al. [99], Barker and Stuckey [16], have in common that they impose a *stratified* use of negation. Because of this, they can refer to the perfect model semantics. As we explained in Section 3.4, in the context of DTM, we cannot expect policies to be stratified. Our approach is thus more powerful than the approaches based on the stratifiable negation. Dung and Thang in [42] propose a DTM system based on logic programming and the *stable model semantics* [50]. They provide a general sufficient condition that guarantees the monotonicity wrt the client submitted credentials. In our work we show that a TM system can be non-monotonic not only in a local setting, but also when the context for negation is known.

3.8 Conclusions

We present the language $RT_{\ominus}$, which adds a construct for 'negation-in-context' to the RT_0 trust management system. We argue the necessity of such a construct and illustrate its use with scenarios from virtual communities which cannot be expressed within the RT framework.

We provide a semantics for $RT_{\ominus}$ by translation to general logic programs. We show that, given the complete policy, the membership relation can be decided by running the translation in systems such as XSB, DLV datalog and Smodels. We also show how, for the case that credentials are issuer traceable [67], the chain discovery algorithm for RT_0 can be extended to $RT_{\ominus}$. We are currently employing $RT_{\ominus}$ to specify virtual community policies in the Freeband project I-SHARE.

In section 3.5 we have assumed that the credentials are issuer traceable and that we are able to obtain all relevant credentials. In our scenario this is realistic; as the coordinators of a virtual community play a central role, they are generally assumed to be available sufficiently often and have sufficient resources to store their own credentials. In general collecting all credentials can be difficult, for example, credentials may be stored elsewhere, entities may be unreachable or messages may be lost. In such a situation, we cannot safely determine that A is not in B's role r by absence of credentials. Instead we could ask B to explicitly state that A is *not* a member of $B.r$. This is sufficient if we know the context of a role (and thus which negative facts we need).

CHAPTER 4

Core TuLiP Logic Programming for Trust Management

An important result of Chapter 2 and Chapter 3 is that when using RT for realistic examples, we need to use different members from the RT family in order to be able to model more sophisticated security policies at different levels of sophistication. For instance, if one needs parameterised roles, one needs to use RT_1. This is however still not sufficient to model threshold or separation of duty policies. To model these, one needs at least RT^T which introduces manifold roles. Then we show that, to model making access control decisions in virtual communities, the whole RT family is not sufficient and we propose $RT_\ominus$ in order to handle this problem. We observe that this variety of dialects makes RT hard to use. From the usability view, one would prefer to use the simplest syntax possible, but, at the same time, having maximum flexibility in order to be able to express the maximum range of different security policies. From the implementation point of view, we regard it as a problem that the type system of RT exists only for the simplest member of the RT family: RT_0.

The goal of this chapter is therefore twofold: (1) we want to have a uniform flexible syntax, and (2) we want to have a storage type system that is at least as good as that of RT_0, but which applies to the whole language. As usual in trust management we use logic programming both for the syntax and as the semantics provider for our trust management language. The problem we need to solve is how to combine the flexibility of logic programming with the decentralisation of RT. Distributed storage is the crucial aspect of decentralised trust management.

To achieve our goals, in this chapter we propose Core TuLiP - the theoretical foundations of a trust management language based on logic programming. We show that Core TuLiP is expressive enough to handle security policies of RT_0, RT_1, RT_2, RT^T, and RT^D (on the other hand, Core TuLiP does not support non-monotonic policies and cannot express policies of $RT_\ominus$). Core TuLiP deals with distributed storage of credentials by the means of a mode system and it enjoys uniform syntax and semantics based on moded logic programming.

The contents of this chapter was first published as M. R. Czenko and S. Etalle. *Core TuLiP - Logic Programming for Trust Management*. In *Proc. 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal*, volume 4670 of *LNCS*, pages 380–394, Berlin, 2007. Springer Verlag.

4.1 Introduction

In Chapter 2 we present one of the most successful TM systems: RT defined by Li, Winsborough and Mitchell [66, 67]. Recall that RT has an LP-based declarative semantics, a syntax which is similar to that of SDSI [36], and offers the possibility of storing credentials either by the *issuer* and/or by the *subject*. The location where the credential is stored is determined by the *type* of the credential. Li et al. show that if all credentials are *well-typed* then there exists a terminating credential chain discovery algorithm which determines whether a given statement is valid in the present state.

Although the RT family is successful in achieving its goals, we believe that the RT family presents drawbacks which are worth investigating and improving. In particular, RT syntax is inflexible to the extent that to accommodate natural things such as separation of duty or thresholds, one has to resort to a number of extensions (RT_1 until RT^D, and RT^T plus $RT_\ominus$ introduced in Chapter 3), which makes the language harder to use and understand. Secondly, while languages from the RT family enjoy a declarative reading, this reading does not reflect the crucial storage information given by the types of RT_0.

One could speculate that to solve these problems one should simply translate each language from the RT family into Logic Programming (as the semantics for the RT family is given in terms of Datalog), and then use the latter to specify and prove authorisation statements. This is however inaccurate, as this translation would lose one of the essential elements that make RT a *trust management* system, in particular the information concerning where credentials should be stored and how credentials can be found when needed.

In this chapter we present *Core TuLiP*, which is the theoretical foundation of the TuLiP (Trust management system based on Logic Programming) system we present in Chapter 6. CoreTuLiP can be seen as a subset of (function-free) moded logic programming, with the essential additional feature that the clauses are not stored at a central authority, but are distributed across the different principals involved in the system. The mode information determines *where* a clause will be stored and a form of *well-modedness* is used to guarantee that, as the computation progresses, enough information is available to *find* the clauses needed to build a proof of the query being evaluated. Since credentials are distributed, CoreTuLiP is not amenable to SLD resolution, and requires a mix of top-down and bottom up reasoning (we already use a similar approach in Chapter 3 when defining the extended credential chain discovery for $RT_\ominus$). Here, we present a terminating algorithm which is able to answer well-moded queries, together with the soundness and completeness result. Finally, we show that RT_0, the core language of the RT family, is basically equivalent to a subset of CoreTuLiP. Doing so, we prove that it is possible to define a true trust management language which is as expressive as RT_0 also in terms of credential distribution without giving up the established LP formalism.

CoreTuLiP, based on LP, has a more flexible underlying syntax than RT, and can be extended in order to increase the expressive power of the language further (we do so in Chapter 6 where we show how Core TuLiP supports user-defined constraints and external constraint evaluation algorithms). Even without any extensions, Core TuLiP already allows one to express threshold and separation of duty policies, which require special additions to RT_0.

The chapter is structured as follows: in Sect. 4.2 we introduce the basics of moded Logic Programming. In Sect. 4.3 we introduce CoreTuLiP. In Sect. 4.4 we present the Lookup

and Inference AlgoRithm, and we show that it is sound and complete w.r.t. the standard LP semantics. In Sect. 4.5 we compare RT_0 with CoreTuLiP. Finally, in Sect. 4.6 we present the related work and then we conclude the chapter and propose future research in Sect. 4.7.

4.2 Preliminaries on Logic Programs

The reader is assumed to be familiar with the terminology and the basic results of the semantics of logic programs [10, 69]. Here, we refer to *function-free* (Datalog-like) logic programs and we adopt the notation of Apt [10]. We denote atoms by $A, B, H, \ldots$, queries by $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ (following the notation used by Apt [10], queries are simply conjunctions of atoms, possibly empty), clauses by $c, d, \ldots$, and programs by P. The empty query is denoted by $\Box$. For any syntactic object (e.g. atom, clause, query) o, we denote by $Var(o)$ the set of variables occurring in o. Given a *substitution* $\sigma = \{x_1/t_1, ..., x_n/t_n\}$ we say that $\{x_1, \ldots, x_n\}$ is its *domain* (denoted by $Dom(\sigma)$) and that $Var(\{t_1, ..., t_n\})$ is its *range* (denoted by $Ran(\sigma)$). Further, we denote by $Var(\sigma) = Dom(\sigma) \cup Ran(\sigma)$. If $t_1, ..., t_n$ is a permutation of $x_1, ..., x_n$ then we say that σ is a *renaming*. The *composition* of substitutions is denoted by juxtaposition ($\theta\sigma(X) = \sigma(\theta(X))$). We say that a syntactic object (e.g. an atom) o is an *instance* of o' iff for some σ, $o = o'\sigma$; o is called a *variant* of o', written $o \approx o'$ iff o and o' are instances of each other. A substitution θ is a *unifier* of objects o and o' iff $o\theta = o'\theta$. We denote by *mgu*(o, o') any *most general unifier* (*mgu*, in short) of o and o'. Computations are sequences of derivation steps. The non-empty query $q : \mathbf{A}, B, \mathbf{C}$ and a clause $c : H \leftarrow \mathbf{B}$ (renamed apart w.r.t. q) yield the resolvent $(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$, provided that $\theta = mgu(B, H)$. A *derivation step* is denoted by $\mathbf{A}, B, \mathbf{C} \overset{\theta}{\Longrightarrow}_{P,c} (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$ where c is called its *input clause*, and B is called the *selected atom* of q. A derivation is obtained by iterating derivation steps. A maximal sequence $\delta := \mathbf{B}_0 \overset{\theta_1}{\Longrightarrow}_{P,c_1} \mathbf{B}_1 \overset{\theta_2}{\Longrightarrow}_{P,c_2} \cdots \mathbf{B}_n \overset{\theta_{n+1}}{\Longrightarrow}_{P,c_{n+1}} \mathbf{B}_{n+1} \cdots$ of derivation steps is called an *SLD derivation of* $P \cup \{\mathbf{B}_0\}$ provided that for every step the standardisation apart condition holds, i.e. the input clause employed at each step is variable disjoint from the initial query $\mathbf{B}_0$, and from the substitutions and the input clauses used at earlier steps. If δ is maximal and ends with the empty query ($\mathbf{B}_n = \Box$) then the restriction of θ to the variables of $\mathbf{B}$ is called its *computed answer substitution* (*c.a.s.*, for short).

Moded Programs. Informally speaking, a *mode* indicates how the arguments of a relation should be used, i.e. which are the input and which are the output positions of each atom, and allows one to derive properties such as absence of run-time errors for Prolog built-ins and absence of floundering for programs with negation [12]. Most compilers encourage the user to specify a mode declaration.

Definition 4.2.1 (Mode) *Consider an n-ary predicate symbol p. By a* mode *for p we mean a function m_p from $\{1, \ldots, n\}$ to $\{In, Out\}$.*

If $m_p(i) = In$ (resp. *Out*), we say that i is an *input* (resp. *output*) *position of* p (with respect to m_p). We assume that each predicate symbol has a *unique mode* associated to it; multiple modes may be obtained by renaming the predicates. We use the notation $(X_1, \ldots, X_n)$ to indicate the mode m in which $m(i) = X_i$. For instance, (In, Out) indicates the mode in which the first (resp. second) position is an input (resp. output) position. To benefit from

the advantage of modes, programs are required to be *well-moded* [12]: they have to respect some correctness conditions relating the input arguments to the output ones. We denote by $In(A)$ (resp. $Out(A)$) the sequence of terms filling in the input (resp. output) positions of A, and by $VarIn(A)$ (resp. $VarOut(A)$) the set of variables occupying the input (resp. output) positions of A.

Definition 4.2.2 (Well-Moded) *A clause* $H \longleftarrow B_1, \ldots, B_n$ *is* well-moded *if* $\forall\, i \in [1, n]$

$$\begin{aligned} VarIn(B_i) &\subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H), \text{ and} \\ VarOut(H) &\subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H). \end{aligned}$$

A query $\mathbf{A}$ *is* well-moded *iff the clause* $H \longleftarrow \mathbf{A}$ *is well-moded, where* H *is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following Lemma, due to [11], shows the "persistence" of the notion of well-modedness.

Lemma 4.2.3 *An SLD-resolvent of a well-moded query and a well-moded clause that is variable-disjoint with this query, is well-moded.* □

As a consequence of Lemma 4.2.3 we have the following well-known property [11]:

Corollary 4.2.4 *Let* P *be a well-moded program and* $\mathbf{A}$ *be a well-moded query. Then for every computed answer* σ *of* $\mathbf{A}$ *in* P, $\mathbf{A}\sigma$ *is ground.* □

A straightforward consequence of this Corollary is the following one:

Corollary 4.2.5 *Let* $H \longleftarrow B_1, \ldots, B_n$ *be a clause in a well-moded program* P. *If* A *is a well-moded atom such that* $\gamma_0 = mgu(A, H)$ *and for every* $i \in [1, j], j \in [1, n-1]$ *there exists a successful derivation* $B_i\gamma_0, \ldots, \gamma_{i-1} \xrightarrow{\gamma_i}_P \Box$, *then* $B_{j+1}\gamma_0, \ldots, \gamma_j$ *is a well-moded atom.* □

4.3 Core TuLiP

We now introduce CoreTuLiP, which to a first approximation is a variant of moded LP. In CoreTuLiP, there are two disjoint types of predicates: (user-defined) *credential predicates* and built-in *constraint predicates*. In CoreTuLiP,

- a credential predicate has arity two;
- in an atom with a credential predicate, we call the term filling in the first argument position the *issuer*, and the one filling in the second argument position the *subject*.
- a *credential* is a clause defining a credential predicate. Then, the *issuer* of a credential is the term filling in the first argument position of the head.

We postpone the discussion on the constraints till Chapter 6 where we present the full version of the language. In the full version of TuLiP a credential predicate may have more than two arguments and it is also possible to use user defined constraints.

Notation From this Chapter, we strictly follow the logic programming notation, where terms starting with an uppercase letter are reserved for variables. Therefore, from now on, role names and entities are strings of characters starting with a lowercase letter.

The following two examples illustrate the use of Core TuLiP credentials and demonstrate the expressive power of Core TuLiP.

Example 4.1 To access a project document at the University of Twente (UT) one must be either a project member and a Ph.D. student at the UT or at one of the partner universities, or be approved by two different assistant professors from the UT. Jerry and Jeroen are assistant professors at the UT. Jerry states that a project member from one of the partner universities can access the document if she is approved by at least one project member who is also an associate professor at that university. Jeroen approves anyone who is also approved by a project leader at the UT. Sandro is a project leader at the UT. This scenario can be modelled with the following set of credentials.

(1) *access_document*(ut, X) $\longleftarrow$
project_member(ut, X),
prof(ut, A_1),
prof(ut, A_2),
$A_1 \neq A_2$,
approve_access(A_1, X),
approve_access(A_2, X).

(2) *access_document*(ut, X) $\longleftarrow$
project_partner(ut, P),
project_member(P, X),
phd_student(P, X).

(3) *approve_access*$(jerry, X)$ $\longleftarrow$
project_partner(ut, P),
project_member(P, X),
associate_prof(P, A),
project_member(P, A),
approve_access(A, X).

(4) *approve_access*$(jeroen, X)$ $\longleftarrow$
project_leader(ut, L),
approve_access(L, X).

(5) *project_leader*$(ut, sandro)$.
(6) *phd_student*$(ut, marcin)$.
(7) *approve_access*$(sandro, rico)$.
(8) *approve_access*$(jeffrey, rico)$.
(9) *associate_prof*$(tud, jeffrey)$.
(10) *project_member*$(ut, jerry)$.
(11) *project_member*$(ut, charles)$.
(12) *prof*$(ut, jerry)$.
(13) *prof*$(ut, jeroen)$.
(14) *project_partner*(ut, ut).
(15) *project_partner*(ut, tud).
(16) *project_member*$(tud, jeffrey)$.
(17) *project_member*$(tud, rico)$.

In the example we see that a credential can represent a role assignment or directly a permission. For instance, in credential (6) *ut* assigns role *phd_student* to *marcin*. On the other hand, credential (1) represents a permission to access a document at the University of Twente. We see that a credential can be a simple logical *fact* (credentials (5)-(17)), or a more sophisticated *rule* (credentials (1)-(4)). A "fact" credential is a direct equivalent of the RT_0 *Type-1* credential (Simple Member). Thus, credentials (5)-(17) can be directly translated into RT_0 credentials. For instance, the RT_0 equivalent of credential (15) is: *ut.project_partner* $\longleftarrow$ *tud*. Credential (4) is very similar to the RT_0 *Type-3* credential (Linking Inclusion), but it is more general in that the entity occurring in the body of the credential (*ut*) can be different from the issuer of the credential (*jeroen*). In order to express credentials (2) and (3) in RT_0 one would have to introduce intermediate roles. For instance, credential (3) can be represented by the following set of RT_0 credentials (here <*project_partner*> shall be replaced by each *project partner* of *ut* - *ut* and *tud* in our setting):

jerry.approve_access $\longleftarrow$ *ut.project_partner.approve_access*
<project_partner>.approve_access $\longleftarrow$ *<project_partner>.project_member* $\cap$
<project_partner>.member_prof.approve_access
<project_partner>.member_prof $\longleftarrow$ *<project_partner>.project_member* $\cap$
<project_partner>.associate_prof

Credential (1) cannot be expressed in RT_0, neither can (1) be expressed in RT_1 or in RT_2. This is because the RT framework does not support constraints to the extent that constraints are supported in Core TuLiP. In order to model a statement like "two different assistant professors from the UT must approve the access to the project document" one needs to use at least RT^{T}. We show this in the the next example.

Example 4.2 Core TuLiP is already expressive enough to express *threshold* and *separation of duty* policies. Modelling a threshold or a separation of duty policy using languages from the RT family requires the adoption of special operators (which are present in more expressive members of the RT family RT_1, RT_2, RT^{T}, or RT^{D}). Consider for instance the following statement presented by Li et al. [66]: "a says that an entity is a member of $a.r$ if one member of $a.r_1$ and two *different* members of $a.r_2$ all say so". This policy cannot be expressed in RT_0, and to express this in RT one needs to use the *manifold roles*, which extend the notion of roles by allowing role members to be *collections* of entities (rather than just principals). This is done in RT^T by defining the operators $\odot$ and $\otimes$. A *type-5* credential of the form $a.r \longleftarrow b_1.r_1 \odot b_2.r_2$ says that $\{s_1 \cup s_2\}$ is a member of $a.r$ if s_1 is a member of $b_1.r_1$ and s_2 is a member of $b_2.r_2$. A *type-6* credential $a.r \longleftarrow b_1.r_1 \otimes b_2.r_2$ has a similar meaning, but it additionally requires that $s_1 \cap s_2 = \emptyset$. With these two additional types, one can express the above statement using the following three credentials:

$$\begin{aligned} a.r &\longleftarrow a.r_4.r \\ a.r_4 &\longleftarrow a.r_1 \odot a.r_3 \\ a.r_3 &\longleftarrow a.r_2 \otimes a.r_2 \end{aligned}$$

In Core TuLiP, on the other hand, this policy can be expressed with the following single credential:

$$r(a, X) :- r_1(a, Y),\ r(Y, X),\ r_2(a, Z_1),\ r_2(a, Z_2),\ Z_1 \neq Z_2,\ r(Z_1, X),\ r(Z_2, X).$$

Notably, to express this, we don't have to use manifold-like structures like the ones used in the RT family.

In TM, a credential is always *issued* by some authority (for the sake of simplicity here we identify authorities with the set of ground terms). In Example 4.1, the credential *prof*(*ut*, *jerry*) is *issued* by *ut* (University of Twente), and has *jerry* as the subject. With this credential, *ut* states that *jerry* is one of the professors at the University of Twente. In a practical setting, this credential is signed by *ut*, and *ut* and *jerry* are placeholders for the implementation dependent identifiers (like public keys or URIs). We discuss the practical issues concerning the deployment of TuLiP in Chapter 5. Because each credential must have an issuer, it is natural to expect that the issuer of a credential should be a ground term.

Definition 4.3.1 *Let* $cl : H \longleftarrow B_1, \ldots, B_n$ *be a clause. We say that* cl *is* well-formed *if it is well-moded and* issuer(H) *is a ground term.*

Modes and Decentralised Storage. A credential predicate may have one of three *legal modes*: (In, In), (In, Out), and (Out, In). The reason why the mode (Out, Out) is considered illegal is that it would allow queries with completely uninstantiated arguments like *prof*(X, Y), in which neither the issuer nor the subject is specified. Unlike in LP, such queries cannot be answered in a TM system because the system does not know where to look for relevant credentials, which could be issued and stored by any authority. By requiring that at least one of the arguments be input, and that the credentials be *traceable* (see Definition 4.3.2 below) we will be able to find the credentials we need to construct the proofs we need.

The salient feature of a trust management system is that credentials are stored in a distributed way. For instance, in Example 4.1, the credential *prof*(*ut*, *john*) which is issued by *ut* could be stored by either *ut* or *john*. Storing it by *john* has the advantage that *john* does not have to fetch the credential at *ut* every time he needs it, which in a highly distributed system may be costly. We call the *depositary* of a credential the authority where the credential is stored. In Core TuLiP, it is the mode of the head of a credential which determines the credential depositary (here, we allow only one mode per relation symbol, so each credential may be stored at one place only; by allowing multiple modes we lift this limitation in the extended system described in Chapter 6). Returning to Example 4.1, if *mode*(*prof*) is either (In, In) or (In, Out), then the credential *prof*(*ut*, *jerry*) will be stored at *ut*, otherwise (if the mode is (Out, In)), *jerry* will store the credential. Storing the credential at some other place would make it unfindable. The definition below generalises this concept.

Definition 4.3.2 (Traceable, Depositary) *We say that a clause* $cl : H \longleftarrow B_1, \ldots, B_n$ *is* traceable *if it is well-formed and one of the following conditions holds:*

1. mode$(H) \in \{(In, In), (In, Out)\}$ *– in this case* issuer(H) *is the* depositary *of the rule,*
2. mode$(H) = (Out, In)$*, and* subject$(H)(= In(H))$ *contains a ground term - in this case* subject(H) *is the* depositary *of the rule,*
3. mode$(H) = (Out, In)$ *and* subject(H) *contains a variable. In this case we require that there exists a prefix* $B_1, \ldots, B_k$ *of the body such that*

   (a) mode$(B_1) = \ldots =$ mode$(B_k) = (Out, In)$,
   (b) $In(H) = In(B_1)$,
   (c) $In(B_{i+1}) = Out(B_i)$*, and is a variable, for* $i \in [1, k-1]$,
   (d) $Out(B_k)$ *contains a ground term,*

   In this case, we say that issuer$(B_k)(= Out(B_k))$ *is the* depositary *of the rule.*

The third case is complex, but it has the advantage of permitting the storage of a credential at a third party (neither the issuer, nor the recipient). Notice, that when the prefix $B_1, \ldots, B_k$ in point 2 in the above definition consists of only one atom (i.e. when $k = 1$) then condition (c) does not apply.

ut	*sandro*	*marcin*	*rico*	*jeffrey*	*jerry*	*tud*
$1, 2, 3, 4, 10, 11, 12, 13, 14$	5	6	$7, 8$	9	–	$15, 16, 17$

Table 4.1: Entities and the credentials stored by each entity as shown in Example 4.3

Example 4.3 Consider the set of credentials from Example 4.1. The University of Twente, the owner of a protected document, wants to have full control over the top-level access control policy for a document. Therefore, UT wants to store credentials (1) and (2) at the UT secured server. UT also decides to store the credentials defining the members of a project and the credentials defining the role *prof*. Jerry and Jeroen, the professors at the UT, both have their own policy for approving access to a university document. The policy of Jerry is given by credential (3) while the policy of Jeroen is given by credential (4). Although Jerry (resp. Jeroen) is the issuer of credential (3) (resp. credential (4)), in order to guarantee that credential (3) (resp. (4)) is always available, Jerry (resp Jeroen) stores the *approve_access* credential at the university server. On the other hand, UT lets Sandro store the credential assigning the role *project_leader* to Sandro (credential (5)) so that Sandro can monitor the requests for this credential. Because there are many Ph.D. students at the university, each Ph.D. student stores her credential. Similarly, each project partner stores the project partnership credential. Finally, Sandro (resp. Jeffrey) trusts Rico to store the *approve_access* credential of which Sandro (resp. Jeffrey) is the issuer - thus Rico stores credentials (7) and (8). Table 4.1 lists the entities and the credentials stored by each entity.

Now, we need to assign a mode to each credential atom so that the mode reflects the desired storage location (the depositary) and guarantees that the credential is traceable (Definition 4.3.2). Table 4.2 shows the credential predicate symbols and the associated mode. Take for example credential (4). From Table 4.2, we have that *mode*(*approve_access*) =

(In, In)	(In, Out)	(Out, In)
access_document *project_member*	*prof*	*approve_access* *project_leader* *phd_student* *associate_prof* *project_partner*

Table 4.2: The predicate symbols and the associated mode in Example 4.3

mode(*project_leader*) = (*Out*,*In*). Now if we look at credential (4) then we see that for this mode assignment the credential is not well-moded, so it is also not well-formed and, as such, not traceable. In order to make credential (4) well-moded and traceable, we need to change the order of the atoms in the body to be as follows:

$$(4)\ \ approve\_access(jeoren, X) \longleftarrow approve\_access(L, X), project\_leader(ut, L).$$

Now, we see that the atoms in the body satisfy the requirements stated in point 3 of Definition 4.3.2. We have to perform a similar action for credential (3). Here, in order to satisfy Definition 4.3.2 we form the prefix in the body (point 3) from the atoms $approve\_access(A, X)$,

associate_prof(P, A), *project_partner*(ut, P), followed by the remaining atoms (in any order):

(3) *approve_access*(*jerry*, X) $\longleftarrow$ *approve_access*(A, X), *associate_prof*(P, A), *project_partner*(ut, P), *project_member*(P, A), *project_member*(P, X).

Finally, in order to make credential (2) well-moded (and by this also traceable) we need to rewrite it as follows:

(2) *access_document*(ut, X) $\longleftarrow$ *phd_student*(P, X), *project_partner*(ut, P), *project_member*(P, X).

Example 4.3 shows that the mode assigned to a credential atom may influence the order of the atoms in the body. This may be counter-intuitive for the user and also makes the reading of a credential more difficult. In Chapter 6 we show how to separate the meaning of the credential from the actual storage given by the mode assignment.

We can now introduce the concept of a state.

Definition 4.3.3 *A* state $\mathcal{P}$ *is a finite collection of pairs* (a, P_a) *where* P_a *is a collection of traceable credentials and* a *is the depositary of these credentials.*

The declarative semantics of a state is simply given in terms of logic programming as follows:constraints are user-defined)

Definition 4.3.4 *Let* $\mathcal{P}$ *be the state* $\{(a_1, P_1), \ldots, (a_n, P_n)\}$*, and* A *be an atom*

- *We denote by* $P(\mathcal{P})$ *the set of clauses* $P_1 \cup \cdots \cup P_n$*. We call* $P(\mathcal{P})$ *the LP-counterpart of state* $\mathcal{P}$*.*
- *We say that* A *is true in state* $\mathcal{P}$ *iff* $P(\mathcal{P}) \cup \mathcal{C} \models A$*, where* $\mathcal{C}$ *is a first order theory determining the meaning of built-in predicates.*

4.4 The Lookup and Inference AlgoRithm (LIAR)

The goal of an authorisation system is to check whether a fact is true in a given state. Since the state $\mathcal{P}$ can be very large and distributed across different agents, it is essential to have an algorithm which takes care of computing whether a given query is true in $\mathcal{P}$ without having to collect the entire $P(\mathcal{P})$. An extra difficulty comes from the fact that clauses might easily be mutually recursive, and that cases 2 and 3 of Definition 4.3.2 make it impossible to follow a straightforward top-down reasoning. In this section we present a suitable algorithm. We begin by giving the necessary definitions.

Definition 4.4.1 (Connected) *We say that two atoms* A *and* B *that have mode* (Out, In) *are* connected *if* subject(A) *is ground and* subject(A) = subject(B).

Let A be an atom and S be a set of atoms. We adopt the following conventions:

(i) We write $A \widetilde{\in} S$ iff $\exists A' \in S$, such that $A' \approx A$ (i.e. A' is a renaming of A).

(ii) We write $A \overset{\sim}{\notin} S$ iff $\nexists A' \in S$ such that $A' \approx A$.

(iii) We write $A \overset{\theta}{\hookrightarrow} S$ iff $\exists A' \overset{\sim}{\in} S$ standardised apart w.r.t. A such that $\gamma = mgu(A, A')$ and $A\theta \approx A\gamma$.

Definition 4.4.2 *Let A be an atomic well-moded query. We define the Lookup and Inference AlgoRithm (LIAR) which given a state $\mathcal{P}$ and a query A as an input returns the (possibly empty) sets of atoms* FACTSTACK *and* GOALSTACK. *The algorithm is reported in Listing 4.1.*

Listing 4.1: The Lookup and Inference AlgoRithm (LIAR). We assume that *dummy* is a reserved predicate symbol, with mode (Out, In). Statements in boxes are optional and included only for optimisation purposes.

INPUT : A. /* A is the initial atomic query */
Init:
  CLSTACK : $\{\square \longleftarrow A\}$;
  FACTSTACK = GOALSTACK = $\boxed{\text{VISITED}} = \emptyset$;
  SATISFIED = **FALSE** ;

REPEAT
  Phase 1 (Top-down resolution):
  CHOOSE:
    c : $H \longleftarrow \mathbf{B}, C, \mathbf{D} \in$ CLSTACK and
    $\mathbf{B}' \subseteq$ FACTSTACK, such that the following conditions hold:
      (i) $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ,
      (ii) $C\theta$ is well-moded,
      (iii)$C\theta \overset{\sim}{\notin}$ GOALSTACK,
      $\boxed{\text{(iv) } \textbf{IF } mode(C) = (Out, In) \textbf{ THEN } subject(C\theta) \notin \text{VISITED } \textbf{ENDIF}}$
  ADD $C\theta$ to GOALSTACK;
  IF *mode*$(C) \in \{(In, Out), (In, In)\}$ **THEN**
    FETCH at *issuer*$(C\theta)$ all clauses $\{c_1, \ldots, c_n\}$ whose head unifies
      with $C\theta$ with mgus $\{\gamma_1, \ldots, \gamma_n\}$ respectively ;
    FOR EACH $c_i\gamma_i \in \{c_1\gamma_1, \ldots, c_n\gamma_n\}$**DO**
      IF $c_i\gamma_i \overset{\sim}{\notin}$ CLSTACK **THEN** ADD $c_i\gamma_i$ to CLSTACK **ENDIF**
    END FOR EACH
  ELSEIF *mode*$(C) = (Out, In)$ **THEN**
    FETCH all clauses $\{c_1, \ldots, c_n\}$ stored at *subject*$(C\theta)$ whose head
      has mode (Out, In) ;
    $\boxed{\text{ADD } \textit{subject}(C\theta) \text{ to VISITED;}}$
    FOR EACH $c_i \in \{c_1, \ldots, c_n\}$ **DO**
      IF $c_i \overset{\sim}{\notin}$ CLSTACK **THEN** ADD c_i to CLSTACK **ENDIF**
    END FOR EACH
  ENDIF
  Phase 2 (Bottom-up model-building):
  REPEAT
    CHOOSE: $H \longleftarrow \mathbf{B} \in$ CLSTACK and $\mathbf{B}' \subseteq$ FACTSTACK,
      such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ ;

IF $H\theta \notin$ FACTSTACK **THEN** ADD $H\theta$ to FACTSTACK **ENDIF**;
IF *mode*$(H) = (Out, In)$ *AND issuer*$(H\theta) \notin$ VISITED **THEN**
ADD to CLSTACK the clause:
$dummy(X, \textit{issuer}(H\theta)) \longleftarrow dummy(X, \textit{issuer}(H\theta))$
where *mode*$(dummy) = (Out, In)$
ENDIF
UNTIL nothing can be added to FACTSTACK;
IF A is ground and $A \in$ FACTSTACK **THEN** SATISFIED $=$ **TRUE ENDIF**
UNTIL SATISFIED **OR** nothing can be added to FACTSTACK and CLSTACK;
OUTPUT $=$ FACTSTACK;

The algorithm maintains three *stacks*: CLSTACK contains the set of clauses collected so far, FACTSTACK contains the set of atomic logical consequences inferred from CLSTACK, and GOALSTACK contains the set of atomic goals already processed (to handle loops). Additionally, the VISITED stack contains the set of entities that have been visited during the processing. Initially, CLSTACK contains a single clause constructed from the initial atomic query A; the other stacks are empty. The algorithm is divided in two phases. Phase 1 contains the credential discovery. First, it selects a *new* well-moded atom $C\theta$ from the body of a clause in CLSTACK and then, depending on its mode, it fetches the *new* credentials from either *issuer*$(C\theta)$ or *subject*$(C\theta)$. By fetching a credential we understand connecting to a remote credential server corresponding to the entity responsible for storing this credential (we discuss the deployment of TuLiP in Chapter 5). The fetched credentials are then added to the CLSTACK. Notice that, when *mode*$(C) = (Out, In)$, *all* clauses whose head has mode (Out, In) must be fetched from *subject*$(C\theta)$, and not only the clauses whose head unifies with $C\theta$. This is because in this case one does not know which credentials may be needed to prove $C\theta$, yet. To overcome this problem, the algorithm overestimates and fetches all credentials with the right mode being stored at *subject*$(C\theta)$. In Phase 2, the model of the set of clauses in the CLSTACK is built bottom-up. Newly inferred facts are added to the FACTSTACK. For the facts having mode (Out, In), the algorithm adds a *dummy* clause to CLSTACK. The mode of the *dummy* predicate symbol is (Out, In) and the second argument of *dummy* is set to *issuer*$(H\theta)$, where $H\theta$ is the newly inferred fact. Later, in Phase 1 all (Out, In) credentials stored at *issuer*$(H\theta)$ will be fetched and added to CLSTACK. The algorithm checks the *issuer*$(H\theta)$ because the issuer of $H\theta$ may store other (Out, In) credentials that can be relevant. As in case of the (Out, In) credentials we do not know which credential are actually needed, the algorithm overestimates and fetches all (Out, In) credentials stored at *issuer*$(H\theta)$. This way "subject traceable" chains can be discovered properly. The algorithm extends naturally to queries containing more than one atom.

In answering a specific query only one instance of the LIAR algorithm is involved. In most typical scenario the algorithm is run by the same entity which issues the query. It is also possible, however, that the entity issuing the query connects to a remote machine on which LIAR is running (possible deployment scenarios for LIAR are discussed in Chapter 5).

Example 4.4 Assume the following state (the credentials shown are the Core TuLiP equivalents of the credentials shown in the RT_0 policy presented in Chapter 2 in Example 2.7):

ePub:

(1) *spdiscount*$(\textit{ePub}, X) \longleftarrow \textit{preferred}(\textit{eOrg}, X), \textit{member}(\textit{acm}, X).$

eOrg:

(2) $\mathit{preferred}(\mathit{eOrg}, X) \longleftarrow \mathit{university}(\mathit{eOrg}, Y), \mathit{student}(Y, X).$

(3) $\mathit{university}(\mathit{eOrg}, X) \longleftarrow \mathit{accredited}(\mathit{abu}, X).$

abu:

(4) $\mathit{accredited}(\mathit{abu}, \mathit{stateU}).$

registrarB:

(5) $\mathit{student}(\mathit{stateU}, X) \longleftarrow \mathit{student}(\mathit{registrarB}, X).$

alice:

(6) $\mathit{student}(\mathit{registrarB}, \mathit{alice}).$

(7) $\mathit{member}(\mathit{acm}, \mathit{alice}).$

Here we show the credentials and their depositaries. We have then that *ePub* stores credential (1), *eOrg* stores credentials (2) and (3), *abu* stores credential (4), *registrarB* stores credential (5), and *alice* stores credentials (6) and (7). For this storage configuration we have the following mode assignment:

$$\begin{aligned} \mathit{mode}(\mathit{spdiscount}) = \mathit{mode}(\mathit{preferred}) &= (\mathit{In}, \mathit{In}), \\ \mathit{mode}(\mathit{university}) = \mathit{mode}(\mathit{accredited}) &= (\mathit{In}, \mathit{Out}), \\ \mathit{mode}(\mathit{student}) = \mathit{mode}(\mathit{member}) &= (\mathit{Out}, \mathit{In}). \end{aligned}$$

We present how LIAR evaluates the goal: $\mathit{spdiscount}(\mathit{ePub}, \mathit{alice})$. In the presentation we show the contents of CLSTACK, GOALSTACK, FACTSTACK, and VISITED as the algorithm progresses.

After initialisation, CLSTACK contains one clause:

CLSTACK :

$(C_1) : \square \longleftarrow \mathit{spdiscount}(\mathit{ePub}, \mathit{alice}).$

GOALSTACK : $\emptyset$.

FACTSTACK : $\emptyset$.

VISITED : $\emptyset$.

The algorithm enters Phase 1 (line 8 in Listing 4.1). Clause (C_1) is the only clause in CLSTACK and is selected with the new goal $C\theta = \mathit{spdiscount}(\mathit{ePub}, \mathit{alice})$. In this case θ is an empty substitution. The algorithm adds $C\theta$ to the GOALSTACK (line 16) and then checks the mode of the selected goal (line 17). Because $\mathit{mode}(\mathit{spdiscount}) = (\mathit{In}, \mathit{In})$ the algorithm knows it should search for the credentials defining *spdiscount* at $\mathit{issuer}(C\theta) = \mathit{ePub}$. *ePub* stores only one credential (credential (1)) whose head unifies with the selected goal with $mgu\ \{X/\mathit{alice}\}$. As the consequence, the following clause is fetched from *ePub* and added to CLSTACK:

$$\mathit{spdiscount}(\mathit{ePub}, \mathit{alice}) \longleftarrow \mathit{preferred}(\mathit{eOrg}, \mathit{alice}), \mathit{member}(\mathit{acm}, \mathit{alice}).$$

The algorithm proceeds to Phase 2 (line 31). As the FACTSTACK is empty and all the clauses in CLSTACK depend on other atoms, no new fact is added to FACTSTACK and the algorithm returns to Phase 1. The contents of the CLSTACK, GOALSTACK, FACTSTACK, and VISITED

is the following:

CLSTACK :

(C_1) $\square \longleftarrow$ *spdiscount*(*ePub*, *alice*).

(C_2) *spdiscount*(*ePub*, *alice*) $\longleftarrow$ *preferred*(*eOrg*, *alice*), *member*(*acm*, *alice*).

GOALSTACK :

(G_1) *spdiscount*(*ePub*, *alice*)

FACTSTACK : $\emptyset$.

VISITED : $\emptyset$.

Notice that all the clauses in the CLSTACK are well-moded. The algorithm chooses clause (C_2) and selects *preferred*(*eOrg*, *alice*) as the new goal. The new goal is then added to GOALSTACK. As *mode*(*preferred*) $= (In, In)$ the algorithm contacts *eOrg* for the credentials matching the selected goal. *eOrg* stores two credentials: credential (2) and credential (3). The head of credential (2) unifies with the selected goal with mgu $\{X/alice\}$. Thus, the following clause is fetched from *eOrg* and added to CLSTACK:

$$preferred(eOrg, alice) \longleftarrow university(eOrg, Y), student(Y, alice).$$

The algorithm moves to Phase 2, but here still nothing new can be added to FACTSTACK, so the algorithm returns to Phase 1. The contents of the CLSTACK, GOALSTACK, and FACTSTACK is the following:

CLSTACK :

(C_1) $\square \longleftarrow$ *spdiscount*(*ePub*, *alice*).

(C_2) *spdiscount*(*ePub*, *alice*) $\longleftarrow$ *preferred*(*eOrg*, *alice*), *member*(*acm*, *alice*).

(C_3) *preferred*(*eOrg*, *alice*) $\longleftarrow$ *university*(*eOrg*, Y), *student*(Y, *alice*).

GOALSTACK :

(G_1) *spdiscount*(*ePub*, *alice*)

(G_1) *preferred*(*eOrg*, *alice*)

FACTSTACK : $\emptyset$.

VISITED : $\emptyset$.

Assume that the clause selected is again clause (C_2). Here the algorithm cannot select the second subgoal, *member*(*acm*, *alice*), of clause (C_2) because the preceding subgoal, *preferred*(*eOrg*,*alice*), is not yet proven. No goal has been selected from clause (C_3) from the CLSTACK yet. The first subgoal in clause (C_3), *university*(*eOrg*, Y), satisfies conditions (i-iv) (lines 12-15) and can be selected as the new goal. Based on the mode associated with the *university* credential predicate (*mode*(*university*) $= (In, Out)$), the algorithm fetches the following clause from *eOrg* and adds to the CLSTACK:

$$university(eOrg, X) \longleftarrow accredited(abu, X).$$

The algorithm enters Phase 2 again, but because we still do not have any facts in CLSTACK, FACTSTACK cannot be extended yet, and algorithm moves back to Phase 1. The contents of the CLSTACK, GOALSTACK, and FACTSTACK is the following:

CLSTACK :
(C_1) $\square \longleftarrow$ *spdiscount*(*ePub*, *alice*).
(C_2) *spdiscount*(*ePub*, *alice*) $\longleftarrow$ *preferred*(*eOrg*, *alice*), *member*(*acm*, *alice*).
(C_3) *preferred*(*eOrg*, *alice*) $\longleftarrow$ *university*(*eOrg*, Y), *student*(Y, *alice*).
(C_4) *university*(*eOrg*, X) $\longleftarrow$ *accredited*(*abu*, X).
GOALSTACK :
(G_1) *spdiscount*(*ePub*, *alice*)
(G_1) *preferred*(*eOrg*, *alice*)
(G_3) *university*(*eOrg*, Y)
FACTSTACK : $\emptyset$.
VISITED : $\emptyset$.

The new goal selected in Phase 1 is *accredited*(*abu*, X) from clause (C_4). This is because there is no subgoal in clauses $(C_1 - C_3)$ that can be selected. Because *mode*(*accredited*) = (In, Out), the algorithm (lines 17-22) goes to *abu* and from there the algorithm fetches the following clause:

accredited(*abu*, *stateU*).

This clause is then added to CLSTACK. The algorithm proceeds to Phase 2. Here, for the first time during this evaluation, CLSTACK contains a simple fact: *accredited*(*abu*, *stateU*). This clause is chosen by the algorithm (lines 33-34) and the atom *accredited*(*abu*, *stateU*) is added to FACTSTACK (line 35). But, now also (C_4) can be selected from CLSTACK for which $\mathbf{B}$ = *accredited*(*abu*, X) unifies with $\mathbf{B}'$ = *accredited*(*abu*, *stateU*) (lines 33-34) with mgu $\theta = \{X/stateU\}$. This means that H = *university*(*eOrg*, X)θ = *university*(*eOrg*, *stateU*) is also added to FACTSTACK. No more new facts can be inferred at the moment and the algorithm returns to Phase 1 with the following contents of the stacks:

CLSTACK :
(C_1) $\square \longleftarrow$ *spdiscount*(*ePub*, *alice*).
(C_2) *spdiscount*(*ePub*, *alice*) $\longleftarrow$ *preferred*(*eOrg*, *alice*), *member*(*acm*, *alice*).
(C_3) *preferred*(*eOrg*, *alice*) $\longleftarrow$ *university*(*eOrg*, Y), *student*(Y, *alice*).
(C_4) *university*(*eOrg*, X) $\longleftarrow$ *accredited*(*abu*, X).
(C_5) *accredited*(*abu*, *stateU*).
GOALSTACK :
(G_1) *spdiscount*(*ePub*, *alice*)
(G_1) *preferred*(*eOrg*, *alice*)
(G_3) *university*(*eOrg*, Y)
(G_4) *accredited*(*abu*, X)
FACTSTACK :
(F_1) *accredited*(*abu*, *stateU*)
(F_2) *university*(*eOrg*, *stateU*)
VISITED : $\emptyset$.

Because *university*$(eOrg, stateU)$ is in FACTSTACK, it means it is proven to be true, and now the algorithm can select the second subgoal from clause (C_3). We have (lines 10-15): $\mathbf{B}$ = *university*$(eOrg, Y)$, $\mathbf{B'}$ = *university*$(eOrg, stateU)$, and $\theta = mgu(\mathbf{B}, \mathbf{B'}) = \{Y/stateU\}$. Therefore, the new goal is *student*$(Y, alice)\theta$ = *student*$(stateU, alice)$. Because *mode*$(student) = (Out, In)$ the algorithm knows that it should search for the related credentials at *subject*$(student(stateU, alice))$ = *alice* (lines 23-29). Alice stores two credentials that have mode (Out, In): credential (6) and credential (7). Because the algorithm tries to discover the related credentials by starting from the subject, the algorithm does not know which (Out, In) credentials are relevant and which are not. For this reason, the algorithm fetches all credentials that have mode (Out, In) from *alice*. In our example, the credentials fetched from *alice* are credentials (6) and (7). The algorithm adds the fetched credentials to CLSTACK and adds *alice* to VISITED. By adding *alice* to VISITED, the algorithm remembers not to fetch any (Out, In) credentials from *alice* later during answering this query, as all credentials moded (Out, In) have already been fetched.
The algorithm moves to Phase 2 where it selects the newly added clauses from CLSTACK and adds *student*$(registrarB, alice)$ and *member*$(acm, alice)$ to FACTSTACK. Because *mode*$(student)$ = *mode*$(member) = (Out, In)$, the algorithm adds the following two clauses to CLSTACK:

---

dummy$(X, registrarB) \longleftarrow$ *dummy*$(X, registrarB)$.
dummy$(X, acm) \longleftarrow$ *dummy*(X, acm).

No new fact can be generated at this point and the algorithm returns to Phase 1. The contents of the stacks is the following:

CLSTACK :

(C_1) $\square \longleftarrow$ *spdiscount*$(ePub, alice)$.
(C_2) *spdiscount*$(ePub, alice) \longleftarrow$ *preferred*$(eOrg, alice)$, *member*$(acm, alice)$.
(C_3) *preferred*$(eOrg, alice) \longleftarrow$ *university*$(eOrg, Y)$, *student*$(Y, alice)$.
(C_4) *university*$(eOrg, X) \longleftarrow$ *accredited*(abu, X).
(C_5) *accredited*$(abu, stateU)$.
(C_6) *student*$(registrarB, alice)$.
(C_7) *member*$(acm, alice)$.
(C_8) *dummy*$(X, registrarB) \longleftarrow$ *dummy*$(X, registrarB)$.
(C_9) *dummy*$(X, acm) \longleftarrow$ *dummy*(X, acm).

GOALSTACK :

(G_1) *spdiscount*$(ePub, alice)$
(G_1) *preferred*$(eOrg, alice)$
(G_3) *university*$(eOrg, Y)$
(G_4) *accredited*(abu, X)
(G_5) *student*$(stateU, alice)$

FACTSTACK :

(F_1) *accredited*$(abu, stateU)$
(F_2) *university*$(eOrg, stateU)$

(F_3) *student*(*registrarB*, *alice*).
(F_4) *member*(*acm*, *alice*).

VISITED :
alice

The algorithm selects *dummy*$(X,$ *registrarB*$)$ as the new goal. The mode of *dummy* is (Out, In) and the algorithm fetches all credentials moded (Out, In) from *registrarB*. As a consequence, credential (5) is fetched and added to CLSTACK and *registrarB* is added to VISITED. The algorithm moves to Phase 2.

In Phase 2, the algorithm selects the newly added clause *student*$($*stateU*$, X) \longleftarrow$ *student*$($*registrarB*$, X)$. from CLSTACK for which *student*$($*registrarB*$, X)$ unifies with *student*$($*registrarB*, *alice*$)$. As the result, *student*$($*stateU*, *alice*$)$ is added to FACTSTACK and also *dummy*$(X,$ *stateU*$) \longleftarrow$ *dummy*$(X,$ *stateU*$)$. is added to CLSTACK. Because *university*$($ *eOrg*, *stateU*$)$ and *student*$($*stateU*, *alice*$)$ are both in FACTSTACK, also *preferred*$($*eOrg*, *alice*$)$ is added to FACTSTACK. This in turn triggers addition of *spdiscount*$($*ePub*, *alice*$)$ to FACTSTACK. Because initial goal $(=$ *spdiscount*$($*ePub*, *alice*$))$ is ground and *spdiscount*$($ *ePub*, *alice*$)$ is in FACTSTACK, the algorithm sets SATISFIED $=$ **TRUE** and the algorithm finishes (lines 42-43) with the following contents of the stacks:

CLSTACK :
(C_1) $\square \longleftarrow$ *spdiscount*(*ePub*, *alice*).
(C_2) *spdiscount*(*ePub*, *alice*) $\longleftarrow$ *preferred*(*eOrg*, *alice*), *member*(*acm*, *alice*).
(C_3) *preferred*(*eOrg*, *alice*) $\longleftarrow$ *university*(*eOrg*, Y), *student*(Y, *alice*).
(C_4) *university*(*eOrg*, X) $\longleftarrow$ *accredited*(*abu*, X).
(C_5) *accredited*(*abu*, *stateU*).
(C_6) *student*(*registrarB*, *alice*).
(C_7) *member*(*acm*, *alice*).
(C_8) *dummy*(X, *registrarB*) $\longleftarrow$ *dummy*(X, *registrarB*).
(C_9) *dummy*(X, *acm*) $\longleftarrow$ *dummy*(X, *acm*).
(C_{10}) *student*(*stateU*, X) $\longleftarrow$ *student*(*registrarB*, X).
(C_{11}) *dummy*(X, *stateU*) $\longleftarrow$ *dummy*(X, *stateU*).

GOALSTACK :
(G_1) *spdiscount*(*ePub*, *alice*)
(G_1) *preferred*(*eOrg*, *alice*)
(G_3) *university*(*eOrg*, Y)
(G_4) *accredited*(*abu*, X)
(G_5) *student*(*stateU*, *alice*)
(G_6) *dummy*(X, *registrarB*)

FACTSTACK :
(F_1) *accredited*(*abu*, *stateU*)
(F_2) *university*(*eOrg*, *stateU*)
(F_3) *student*(*registrarB*, *alice*)

(F_4) *member*(*acm*, *alice*)
(F_5) *student*(*stateU*, *alice*)
(F_6) *preferred*(*eOrg*, *alice*)
(F_7) *spdiscount*(*ePub*, *alice*)

VISITED :
alice
registrarB

The following results show that LIAR algorithm is sound and complete w.r.t. the standard LP semantics, i.e. the centralised algorithm based on the SLD resolution. We need the following lemma.

Lemma 4.4.3 *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of the algorithm execution for some well-moded query. Let A be an atom in* FACTSTACK. *Then A is ground.*

Proof. The proof proceeds by induction on the size of FACTSTACK. In the basic case FACTSTACK is empty and so the proposition automatically holds.
Now, assume that FACTSTACK contains only ground atoms. We are proving that each time a new atom is added to FACTSTACK, it is ground. Notice that an atom is added to FACTSTACK as the result of the bottom-up evaluation of the facts in FACTSTACK and a clause selected from CLSTACK. We have then two cases: (1) the clause selected from CLSTACK in Phase 2 of the algorithm has an empty body, (2) the clause selected from CLSTACK in Phase 2 of the algorithm has a non-empty body.
Case 1: The clause selected from CLSTACK has an empty body.
In such a case, a fact can be added to FACTSTACK only if it is already in CLSTACK. Let $H.$ be a clause selected from CLSTACK. Recall that $\forall C \in$ GOALSTACK, C is well-moded.

1. *mode*$(H) \in \{(In, In), (In, Out)\}$.

   If $H. \in$ CLSTACK then there must be some C in GOALSTACK, such that $\exists\theta = mgu(H, C)$ and $\exists H' \leftarrow . \in \mathcal{P}$ such that $H' \leftarrow .$ is stored at *issuer*(C), $H = H'\theta$, and $\theta = mgu(H', C)$. But, by Definition 4.3.4 (State), all clauses in a state $\mathcal{P}$ are traceable, so that $\forall G \in$ GOALSTACK and $\forall A \leftarrow . \in \mathcal{P}$ such that *mode*$(G) \in \{(In, In), (In, Out)\}$, $A \leftarrow .$ is stored at *issuer*(G), and $\exists\gamma = mgu(A, G)$, $A\gamma$ is ground and will be added to CLSTACK during Phase 1 of the algorithm. Then, as a special case of the observation above, H must be ground.

2. *mode*$(H) = (Out, In)$.

   If $H\leftarrow. \in$ CLSTACK then there must be some C in GOALSTACK such that *mode*$(C) = (Out, In)$, $\exists c \in \mathcal{P}$ such that c is stored at *subject*(C), and $H\leftarrow. = c$. But, by Definition 4.3.2 (Traceable, Depositary) and by the fact that every traceable clause is well-formed, $\forall G \in$ GOALSTACK such that *mode*$(G) = (Out, In)$ and $\forall D\leftarrow. \in \mathcal{P}$ such that D is connected to C, $D\leftarrow.$ is ground. Then, as a special case, $H\leftarrow.$ must also be ground.

Case 2: The clause selected from CLSTACK has a non-empty body.

When the clause selected from CLSTACK has a non-empty body, a fact can be added to FACTSTACK only be the means of the bottom-up evaluation in Phase 2 of the algorithm. Let $c : H \longleftarrow \mathbf{B}$ be a clause selected from CLSTACK.

1. *mode*$(H) \in \{(In, In), (In, Out)\}$.

   In such a case each input position in the head of c is ground because before c was added to CLSTACK, $head(c)$ was unified with a well-moded atom from GOALSTACK. By well-modedness of clauses, each variable V in the output position of the head of c, such that $V \notin VarIn(H)$, must occur in $\mathbf{B}$. Now, assume that $\exists \mathbf{B}' \subseteq$ FACTSTACK such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ and that $H\theta$ is not ground. Then, it must be that $\exists B \in \mathbf{B}'$ such that B is not ground. But, by the inductive hypothesis, each $B \in \mathbf{B}'$ is ground. This is a contradiction so $H\theta$ must be ground.

2. *mode*$(H) = (Out, In)$.

   If *mode*$(H) = (Out, In)$, then by Definition 4.3.1 (Well-Formed) $Out(H)$ is ground and by Definition 4.3.2 (Traceable,Depositary) either $In(H)$ is ground, or $In(H)$ is a variable and $In(H) = In(B_1)$ where B_1 is the first atom in $\mathbf{B}$. Now, assume that $\exists \mathbf{B}' \subseteq$ FACTSTACK such that $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ and that $H\theta$ is not ground. Then, it must be that $\exists B \in \mathbf{B}'$ such that B is not ground. But, by the inductive hypothesis, each $B \in \mathbf{B}'$ is ground. This is a contradiction so $H\theta$ must be ground.

□

The soundness result is a direct consequence of the construction of the algorithm.

Theorem 4.4.4 (soundness) *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of executing LIAR on $\mathcal{P}$ and a well-moded query. Then $\forall A \in$ FACTSTACK, $P(\mathcal{P}) \models A$.*

Proof. It is easy to see that, by construction, if an atom A is added to FACTSTACK, then CLSTACK $\models A$. Since $\forall c \in$ CLSTACK c is an instance of a clause $c' \in P(\mathcal{P})$, it follows that $P(\mathcal{P}) \models A$. □

The following completeness result guarantees that, after executing LIAR on a state $\mathcal{P}$ and some well-moded query, for any goal $A \in$ GOALSTACK it holds that if there exists a successful SLD derivation of A in $P(\mathcal{P})$ with c.a.s. θ then $A \overset{\theta}{\hookrightarrow}$ FACTSTACK.

Theorem 4.4.5 (completeness) *Let $\mathcal{P}$ be a state and* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal.*

Then $\forall C \in$ GOALSTACK, if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$ FACTSTACK.

Proof. We prove a more general proposition:

Proposition 4.4.6 *Let $\mathcal{P}$ be a state and let* FACTSTACK, GOALSTACK *be the result of executing LIAR on $\mathcal{P}$ and a given well-moded goal. Then $\forall C \in$ GOALSTACK:*

1. *if* mode$(C) = (In, Out)$ *or* mode$(C) = (In, In)$ *and $\exists$ successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$* FACTSTACK,

2. *if* mode$(C) = (Out, In)$ *and $\exists$ successful SLD derivation $D \xrightarrow{\theta}_{P(\mathcal{P})} \square$, where D is an atom connected to C then $D \overset{\theta}{\hookrightarrow}$* FACTSTACK.

Proof. The proof proceeds by induction on the length of the derivation.
Base case: $length = 1$.

1. Assume that $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ has length 1. In such a case there exists a clause $c : C'\leftarrow . \in P$ such that $mgu(C, C') = \theta$. Note that *mode*$(C) \in \{(In, Out), (In, In)\}$. This means that clause c is stored at *issuer*(C) (the mode is assigned to the predicate symbol and this is the same for C and C').

   Since $C \in$ GOALSTACK then:

   (a) first, at some point in Phase 1 of the algorithm (lines 17-22), clause c was fetched at *issuer*(C') and $(C'\leftarrow .)\theta$ was added to CLSTACK;
   (b) then, at some point in Phase 2 of the algorithm (lines 33-35), $C'\theta$ was added to FACTSTACK.

   Since $mgu(C, C') = \theta$ the proposition follows.

2. Assume that $\delta : D \xrightarrow{\theta}_{P(\mathcal{P})} \square$ has length 1. Then there exists a clause $d : D'\leftarrow . \in P$ such that $mgu(D, D') = \theta$. Note that since D is a well-moded goal, *subject*$(D) =$ *subject*(D') and because all clauses are assumed to be well-formed then *issuer*(D') is ground and it holds that *issuer*$(D') =$ *issuer*(D). Since D is connected to C, *mode*$(D) = (Out, In)$ and d is stored at *subject*$(C) =$ *subject*(D). Consequently, since $C \in$ GOALSTACK:

   (a) at some point in Phase 1 of the algorithm (lines 23-29) clause d was fetched at *subject*(D') and added to CLSTACK;
   (b) then, at some point in Phase 2 (lines 33-35), D' was added to FACTSTACK.

   Since $mgu(D, D') = \theta$ the proposition follows.

Inductive case:

1. Assume that there exists an SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$, such that $length(\delta) = m > 1$. Then, by the variant corollary [9], there exists a clause $c : H \longleftarrow B_1, \ldots, B_n$ and substitutions $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that:

   - $\gamma_0 = mgu(C, H)$,
   - $\forall i \in [1, n]$ there exists a successful derivation $\delta_i : B_i\gamma_0 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ such that $length(\delta_i) < m$ with *c.a.s.* γ_i,
   - $C\theta$ is a variant of $H\gamma_0\gamma_1 \cdots \gamma_n$.

   Since $mode(C) \in \{(In, Out), (In, In)\}$ then clause c is stored at *issuer*(C), and, since $C \in$ GOALSTACK at some step in Phase 1 of the algorithm (lines 17-22), c is fetched at *issuer*(C) and $c\gamma_0$ is added to CLSTACK. We need to prove the following claim:

   Claim 1 *For each* $i \in [1, n]$, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.

   Proof of Claim 1. The proof is by induction on i:

- *Basic case*: $i = 1$.
  Because clause c is well-moded, $B_1\gamma_0$ is well-moded. Since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK.
- *Inductive case*:
  Assume $(B_1, \ldots, B_{i-1})\gamma_0 \overset{\gamma_1 \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. By Corollary 4.2.5, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK.
  Now, by Corollary 4.2.4, $B_i\gamma_0, \ldots, \gamma_i$ is ground. By composing the substitutions, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. $\square$

From Claim 1 it follows that, at some point of Phase 2 of the algorithm (lines 33-35), $H\gamma_0\gamma_1 \cdots \gamma_n$ was added to FACTSTACK. Since $\gamma_0 = mgu(C, H)$, it follows that $C \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$.

2. Assume that there exists a successful SLD derivation $\delta : D \xrightarrow{\theta}_{P(\mathcal{P})} \square$, such that $length(\delta) = m > 1$. Then, by the variant corollary [9], there exists a clause $c : H \longleftarrow B_1, \ldots, B_n$ and substitutions $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that:

   - $\gamma_0 = mgu(D, H)$,
   - $\forall i \in [1, n]$ there exists a successful derivation $\delta_i : B_i\gamma_0 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ such that $length(\delta_i) < m$ with *c.a.s.* γ_i,
   - $D\theta$ is a variant of $H\gamma_0\gamma_1 \cdots \gamma_n$.

   Since $mode(D) = (Out, In)$ then either:

   (2a) $In(H)$ contains a ground term a and c is stored at a, *or*

   (2b) $In(H)$ is a variable. In such a case, there exists a prefix $B_1, \ldots, B_k$ of $B_1, \ldots, B_n$ satisfying the conditions of Definition 4.3.2, and c is stored at *issuer*$(B_k) = Out(B_k)$.

   Case 2a.

   Since $C \in$ GOALSTACK and *subject*(C) = *subject*(H) = *subject*(D) then, at some point of Phase 1 of the algorithm (lines 23-29), c is fetched at *subject*(C) and added to CLSTACK. We need to prove the following claim:

   Claim 2 *For each* $i \in [1, n]$, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.

   Proof of Claim 2. The proof is by induction on i:

   - *Basic case*: $i = 1$.
     Because c is well-moded, by Lemma 4.2.3, $B_1\gamma_0$ is also well-moded. Since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK.

- *Inductive case*:
  Assume $(B_1, \ldots, B_{i-1})\gamma_0 \overset{\gamma_1 \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. By Corollary 4.2.5, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK.
  By composing the substitutions, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. □

Now, from Claim 2 and the fact that $c \in P$, it follows that, at some point in Phase 2 of the algorithm (lines 33-35), $H\gamma_0\gamma_1 \cdots \gamma_n$ was added to FACTSTACK. Since $\gamma_0 = mgu(D, H)$, it follows that $D \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$. □

Case 2b.

We first prove the following claim:

Claim 3 *For* $i \in [1, k]$, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.

Proof of Claim 3. In the proof of Claim 3 we also show that $(B_1, \ldots, B_k)\gamma_0 \overset{\gamma_1 \cdots \gamma_k}{\hookrightarrow}$ FACTSTACK implies that, at some point, the algorithm fetches clause c from its depositary and adds it to CLSTACK. The proof is by induction on i:

- *Basic case*: $i = 1$.
  By Lemma 4.2.3, $B_1\gamma_0$ is well-moded. Also *mode*$(B_1) = (Out, In)$, and *subject*(B_1) = *subject*(C) = *subject*(D) and is ground. Since $C \in$ GOALSTACK, and since $B_1\gamma_0 \xrightarrow{\gamma_1}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation, $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK. Since $B_1\gamma_0 \overset{\gamma_1}{\hookrightarrow}$ FACTSTACK then at some point in Phase 2 of the algorithm (lines 36-40) the following *dummy* clause is added to CLSTACK:

  $$dm : dummy(X, \textit{issuer}(B_1\gamma_0\gamma_1)) \text{ :- } dummy(X, \textit{issuer}(B_1\gamma_0\gamma_1)).$$

  Then, at some point in Phase 1 of the algorithm (lines 9-16), dm was selected from CLSTACK and $dummy(X, \textit{issuer}(B_1\gamma_0\gamma_1))$ was added to GOALSTACK. Because *mode*$(dummy) = (Out, In)$, in lines 23-29 *all* clauses moded (Out, In) were fetched from *issuer*$(B_1\gamma_0\gamma_1)$. But, by Definition 4.3.2, *issuer*$(B_1\gamma_0\gamma_1)$ is the depositary of clause c, so also clause c was fetched and added to CLSTACK.
- *Inductive case*:
  Assume $(B_1, \ldots, B_{i-1})\gamma_0 \overset{\gamma_1 \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. By Corollary 4.2.5, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1}$ is well-moded. Since, by inductive hypothesis on the length of the derivation, $B_{i-1}\gamma_0 \cdots \gamma_{i-2} \overset{\gamma_{i-1}}{\hookrightarrow}$ FACTSTACK, at some point in Phase 2 of the algorithm (lines 36-40), the following $dummy$ clause was added to CLSTACK:

  $$dm : dummy(X, \textit{issuer}(B_{i-1}\gamma_0 \cdots \gamma_{i-1})) \text{:-}\ dummy(X, \textit{issuer}(B_{i-1}\gamma_0 \cdots \gamma_{i-1})).$$

Recall that $mode(dummy) = (Out, In)$ (line 39) and that dm is well-moded ($issuer(B_{i-1}\gamma_0 \cdots \gamma_{i-1})$ is a ground term). Since, by Definition 4.3.2, for any $i \in [1, k-1]$, $In(B_{i+1}) = Out(B_i)$, then $B_i\gamma_0 \cdots \gamma_{i-1}$ is connected to $dummy(X, issuer(B_{i-1}\gamma_0 \cdots \gamma_{i-1}))$. This implies that at some point in Phase 1 of the algorithm all the clauses moded (Out, In) were fetched at $subject(B_i\gamma_0 \cdots \gamma_{i-1})$ and added to CLSTACK.

Now, since $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0\gamma_1 \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK. By composing the substitutions, $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, and the claim follows. □

As an immediate consequence of Claim 3, at some point in Phase 1 of the algorithm (lines 23-29), all the clauses moded (Out, In) from $issuer(B_k\gamma_0 \cdots \gamma_k)$ were added to CLSTACK. In particular, clause c was added to CLSTACK.

For the remaining atoms of the body we prove the following claim.

Claim 4 *For $i \in [k+1, n]$, $(B_{k+1}, \ldots, B_i)\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1} \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK.*

Proof of Claim 4. The proof is again by induction on i:

- *Basic case*: $i = k + 1$.
  Notice that $B_{k+1}\gamma_0 \cdots \gamma_k$ is well-moded. Since $B_{k+1}\gamma_0 \cdots \gamma_k \xrightarrow{\gamma_{k+1}}_{P(\mathcal{P})} \square$ is a derivation of $length < m$, by inductive hypothesis on the length of the derivation $B_{k+1}\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1}}{\hookrightarrow}$ FACTSTACK.
- *Inductive case*:
  Assume $(B_{k+1}, \ldots, B_{i-1})\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1} \cdots \gamma_{i-1}}{\hookrightarrow}$ FACTSTACK. Notice again that, by Corollary 4.2.4, $B_i\gamma_0 \cdots \gamma_k\gamma_{k+1} \cdots \gamma_{i-1}$ is well-moded. Since $B_i\gamma_0 \cdots \gamma_k\gamma_{k+1} \cdots \gamma_{i-1} \xrightarrow{\gamma_i}_{P(\mathcal{P})} \square$ is a derivation of $length < m$ then, by inductive hypothesis on the length of the derivation, $B_i\gamma_0 \cdots \gamma_k \cdots \gamma_{i-1} \overset{\gamma_i}{\hookrightarrow}$ FACTSTACK. By composing the substitutions, $(B_{k+1}, \ldots, B_i)\gamma_0 \cdots \gamma_k \overset{\gamma_{k+1} \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK, so that the claim follows. □

From Claim 3 and Claim 4 it follows that for $i \in [1, n]$ $(B_1, \ldots, B_i)\gamma_0 \overset{\gamma_1 \cdots \gamma_i}{\hookrightarrow}$ FACTSTACK. From this and from the fact that $c \in P$ it follows that at some point in Phase 1 of the algorithm (lines 33-35), $H\gamma_0\gamma_1 \cdots \gamma_n$ was added to FACTSTACK. Since $\gamma_0 = mgu(D, H)$, it follows that $D \overset{\theta}{\hookrightarrow}$ FACTSTACK, where $\theta = \gamma_0\gamma_1 \cdots \gamma_n$. □

By observing that each well-moded atom having mode (Out, In) is connected to itself then the completeness result is an immediate consequence of Proposition 4.4.6. □

4.5 Core TuLiP vs. RT_0

In this section we compare Core TuLiP with RT_0 and we show show that Core TuLiP is at least as expressive as RT_0. The RT family is presented in Chapter 2. In order to simplify

RT_0 Storage Type	Core TuLiP Mode
issuer-traces-all	(In, Out)
issuer-traces-def	(In, In)
subject-traces-all	(Out, In)

Table 4.3: RT_0 storage types and Core TuLiP modes

the presentation we assume that each role has just one of the following three type values: *issuer-traces-all (ITA)*, *issuer-traces-def (ITD)*, and *subject-traces-all (STA)*. To extend the results presented in this section to the full version (i.e. including all possible combinations of RT_0 types), in Chapter 6 we extend Core TuLiP by allowing predicates with multiple modes.

4.5.1 Translating RT_0 into Core TuLiP

In this section we demonstrate that Core TuLiP is at least as expressive as RT_0 by showing that an arbitrary RT_0 policy can be translated into an equivalent Core TuLiP state (in Core TuLiP a *state* is an equivalent of a distributed RT_0 policy). First, we define a mapping T from RT_0 to Core TuLiP.

Definition 4.5.1 *Let c be an RT_0 credential. Then $T(c)$ is defined as follows ($ITA = $ issuer-traces-all):*

$$
\begin{aligned}
T(a.r \longleftarrow d) &= r(a, d). \\
T(a.r \longleftarrow b.r_1) &= r(a, X) :\!- r_1(b, X). \\
T(a.r \longleftarrow a.r_1.r_2) &= \begin{cases} r(a, X) :\!- r_2(Y, X), r_1(a, Y). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a, X) :\!- r_1(a, Y), r_2(Y, X). \textit{ otherwise.} \end{cases} \\
T(a.r \longleftarrow b_1.r_1 \cap b_2.r_2) &= \begin{cases} r(a, X) :\!- r_2(b_2, X), r_1(b_1, X). \textit{ if } type(r_1) \neq \textit{ITA}, \\ r(a, X) :\!- r_1(b_1, X), r_2(b_2, X). \textit{ otherwise.} \end{cases}
\end{aligned}
$$

Concerning the mode of the predicates, Table 4.3 shows an RT_0 storage type and the corresponding Core TuLiP mode. □

Recall that in order to preserve the well-modedness of a credential, sometimes we need to change the order of the atoms in the body of a credential. We observed this already in Example 4.3. For the same reason, in Definition 4.5.1, we have to change the order of the atoms in the body of a credential when storage type is other than *issuer-traces-all*. The following theorem shows that, from the view point of the declarative semantics, $\mathcal{S}$ and $T(\mathcal{S})$ are equivalent (recall that m is the fixed predicate symbol used in $SP(\mathcal{S})$: because we directly compare Core TuLiP with RT_0, here we use the original notation with the $m/3$ predicate symbol to specify the semantics of RT_0).

Theorem 4.5.2 *Let $\mathcal{S}$ be an RT_0 policy. Then $SP(\mathcal{S}) \models m(a, r, d)$ iff $T(\mathcal{S}) \models r(a, d)$.*

Proof. Take an RT statement $a.r \longleftarrow d$. The meaning of this statement is given by the clause $SP(a.r \longleftarrow d) = m(a, r, d)$ (see Chapter 2, Section 2.2.2). The Core TuLiP equivalent of this statement is given by $T(a.r \longleftarrow d) = r(a, d)$. Generalising this, we now define a

mapping *sp_to_tulip* which transforms atoms of the form $m(x, y, z)$ into atoms of the form $y(x, z)$ (the mapping is extended to clauses and programs in a natural way). Now if (1) $m/3$ is the only predicate symbol defined in program P, and if (2) each second argument of each atom occurring in P is ground, then *sp_to_tulip* is only a syntactic mapping, and that for each ground atom A, we have that $P \models A$ iff *sp_to_tulip*$(P) \models$ *sp_to_tulip*(A). Now, since for any $SP(\mathcal{S})$ we have that (1) and (2) are both satisfied, and since *sp_to_tulip*$(SP(\mathcal{S})) = T(\mathcal{S})$, the thesis follows (see also Fig. 4.1). □

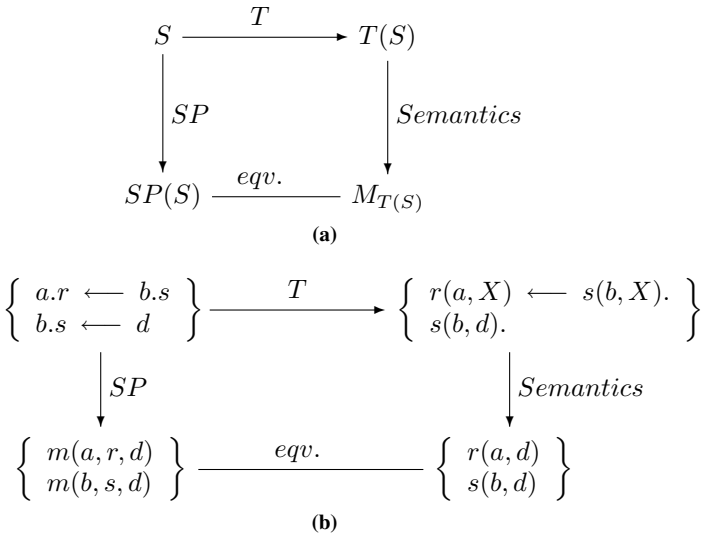


Fig. 4.1: Commutating diagrams illustrating Theorem 4.5.2

Figure 4.1(a) shows the commutating diagram illustrating Theorem 4.5.2 while Fig. 4.1(b) shows an instance of the diagram presented in Fig. 4.1a for a concrete RT_0 policy. In Fig. 4.1(a), $M_{T(\mathcal{S})}$ denotes the (minimal Herbrand) model of program $T(\mathcal{S})$.
Theorem 4.5.2 shows that each RT_0 policy can be translated into a declaratively equivalent Core TuLiP state. Now, to prove the full equivalence we still have to prove two things: that (a) if an RT_0 credential is stored at entity a then the corresponding Core TuLiP statement is stored at a as well, and that (b) the Core TuLiP system is capable of answering the same queries the RT_0 system can.

Proposition 4.5.3 *Let c be an RT_0 credential.*

(a) If c is stored at a then $T(c)$ is also stored at a.

(b) If c is a well-typed then $T(c)$ is traceable.

Proof. (a) This is a direct consequence of Definition 4.5.1 and Definition 4.3.2. Concerning *(b)*, Table 4.4 shows all possible well typed RT_0 credentials, their Core TuLiP counterparts, and the corresponding modes. Using Definition 4.3.2 one can check that for each well typed RT_0 credential shown in Table 4.4 the corresponding Core TuLiP clause is traceable. □

Finally, we have to show how RT_0 goals can be transformed into (legal, i.e. well-moded) Core TuLiP queries.

RT_0 credential (c)	Possible types for r, r_1, and r_2 such that the credential is well typed			Translation to Core TuLiP ($T(c)$)	Modes		
	r	r_1	r_2		r	r_1	r_2
$a.r \longleftarrow d$	ITA			$r(a, d).$	(I,O)		
	STA			$r(a, d).$	(O,I)		
	ITD			$r(a, d).$	(I,I)		
$a.r \longleftarrow b.r_1$	ITA	ITA		$r(a, X) :- r_1(b, X).$	(I,O)	(I,O)	
	STA	STA		$r(a, X) :- r_1(b, X).$	(O,I)	(O,I)	
	ITD	ITA		$r(a, X) :- r_1(b, X).$	(I,I)	(I,O)	
	ITD	ITD		$r(a, X) :- r_1(b, X).$	(I,I)	(I,I)	
	ITD	STA		$r(a, X) :- r_1(b, X).$	(I,I)	(O,I)	
$a.r \longleftarrow a.r_1.r_2$	ITA	ITA	ITA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,O)	(I,O)	(I,O)
	STA	STA	STA	$r(a, X) :- r_2(Y, X), r_1(a, Y).$	(O,I)	(O,I)	(O,I)
	ITD	ITA	ITA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(I,O)
	ITD	ITA	ITD	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(I,I)
	ITD	ITA	STA	$r(a, X) :- r_1(a, Y), r_2(Y, X).$	(I,I)	(I,O)	(O,I)
	ITD	ITD	STA	$r(a, X) :- r_2(Y, X), r_1(a, Y).$	(I,I)	(I,I)	(O,I)
	ITD	STA	STA	$r(a, X) :- r_2(Y, X), r_1(a, Y).$	(I,I)	(O,I)	(O,I)
$a.r \longleftarrow b_1.r_1$ $\cap\, b_2.r_2$	ITA	ITA	ITA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(I,O)
	ITA	ITA	ITD	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(I,I)
	ITA	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,O)	(I,O)	(O,I)
	ITA	ITD	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,O)	(I,I)	(I,O)
	ITA	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,O)	(O,I)	(I,O)
	STA	STA	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(O,I)
	STA	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(I,O)
	STA	STA	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(O,I)	(I,I)
	STA	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(O,I)	(I,O)	(O,I)
	STA	ITD	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(O,I)	(I,I)	(O,I)
	ITD	ITA	ITA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(I,O)
	ITD	ITA	ITD	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(I,I)
	ITD	ITA	STA	$r(a, X) :- r_1(b_1, X), r_2(b_2, X).$	(I,I)	(I,O)	(O,I)
	ITD	ITD	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(I,O)
	ITD	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,O)
	ITD	STA	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(O,I)
	ITD	STA	ITA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,O)
	ITD	STA	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(O,I)	(I,I)
	ITD	ITD	STA	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(O,I)
	ITD	ITD	ITD	$r(a, X) :- r_2(b_2, X), r_1(b_1, X).$	(I,I)	(I,I)	(I,I)

Table 4.4: Well typed RT_0 credentials and the corresponding Core TuLiP traceable clauses their modes (I=In, O=Out)

Translating RT_0 goals Let $\mathcal{S}$ be a well-typed RT_0 policy and $T(\mathcal{S})$ its Core TuLiP equivalent. Let us consider the different sorts of goals supported by RT_0 (refer also to Chapter 2, Section 2.3).

Sort 1: the goal of this sort is "given $a.r$ and b, check if b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". Semantically, answering a goal of this sort is equivalent to checking if $SP(\mathcal{S}) \models m(a, r, b)$. In Core TuLiP, this goal is translated into the query $r(a, b)$, which, being ground, is well-moded. Therefore, by Theorem 4.5.2, we have that $SP(\mathcal{S}) \models m(a, r, b)$ *iff* $T(\mathcal{S}) \models r(a, b)$ which means that goals of Sort 1 can be expressed in Core TuLiP.

Sort 2: the goal of this sort is "given $a.r$, list all principals in $[\![a.r]\!]_{SP(\mathcal{S})}$". Semantically, answering a goal of this sort is equivalent to finding all instances of X such that $SP(\mathcal{S}) \models m(a, r, X)$. This goal can be answered in RT_0 only if role r has type *issuer-traces-all*. The Core TuLiP translation of this goal is the query $r(a, X)$, which is well-moded. This is because, the mode of r in $T(\mathcal{S})$ is (In, Out). Therefore, by Theorem 4.5.2, we have that $SP(\mathcal{S}) \models m(a, r, X)$ *iff* $T(\mathcal{S}) \models r(a, X)$ which means that goals of Sort 2 can be expressed in Core TuLiP.

Sort 3: the goal of this sort is "given b, list all $a.r$ such that b is a member of $[\![a.r]\!]_{SP(\mathcal{S})}$". Semantically, answering a goal of this sort is equivalent to finding all instances of X and Y such that $SP(\mathcal{S}) \models m(X, Y, b)$. In Core TuLiP, we decided not to support this sort of a goal. A goal of Sort 3 can be answered only for the subject traceable role, which means that, in general, a goal of Sort 3 cannot be answered completely. For this reason, we do not distinguish a goal of Sort 3 as a valid goal in Core TuLiP. In RT_0, a goal of Sort 3 is also not used on its own, but only as a subgoal in the goals of Sort 1. Formally eliminating goals of Sort 3 from Core TuLiP lets us to keep the syntax manageable (to express goals of this sort we would need a "polymorphic" mode system in which the actual mode of an atom does not only depend on its predicate symbol but also on some of its arguments). Actually, our LIAR algorithm would be able to answer such goals as well.

Summarising, Theorem 4.5.2 and Proposition 4.5.3 allow us to say that Core TuLiP is at least as expressive as RT_0 (with the restriction of the goals of Sort 3 which are not supported in Core TuLiP - see Section 4.5.1 for details).

4.6 Related Work

We present the related work on Trust Management in Chapter 2. In most of the approaches reported in Chapter 2 [25, 27, 44, 87], it is assumed that all required credentials can be found when needed - the problem of the distributed storage thus is not considered. The exception is RT which provides a type system to deal with the problem of distributed storage. In Sect. 4.5 we compare the type system of RT_0 with our approach to the credential discovery problem.

From other approaches not mentioned in Chapter 2, Li, Grosof, and Feigenbaum [64] develop a logic-based language, called Delegation Logic (DL), to represent policies, credentials, and requests for distributed authorisations. The monotonic version of Delegation Logic – called D1LP – is based on Logic Programming language Datalog. D1LP extends Datalog with constructs that allow specification of the delegation depth and a wide variety of complex principals. Similarly to the approaches mentioned above, Delegation Logic does not deal with distributed storage of credentials.

Bertino et al. [22] introduce an XML-based trust negotiation language $\mathcal{X}$-TNL used

for expressing credentials and disclosure policies in the *Trust-$\mathcal{X}$* system. Though Trust-$\mathcal{X}$ policies are formalised using logic rules, it also does not directly support credential discovery.

PeerTrust [74] is a Trust Negotiation language where the problem of the distributed storage is also taken into account. PeerTrust is based on first order Horn clauses of the form $lit_0 \longleftarrow lit_1, \ldots, lit_n$, where each lit_i is a positive literal. PeerTrust supports distributed storage by allowing each literal in a rule to have an additional *Issuer* argument: lit_i @ *Issuer*. The *Issuer* is the peer responsible for evaluating lit_i. Even though the *Issuer* argument indicates who is responsible for evaluating the credential, the *Issuer* argument does not say where a particular credential should be stored. This means that PeerTrust makes a silent assumption about the credentials being stored in such a way that *Issuer* can find the proof, but PeerTrust gives no clue of how this should be done.

PeerAccess [102] addresses the problem of distributed credential discovery in a slightly different way. In PeerAccess, if an entity needs to establish the truth value of a formula, and is unable to do so on her own, she can ask other entities in the system for help. PeerAccess employs so called *proof hints* to restrict the number of entities that should be contacted for missing information. This approach gives flexibility (one may use proof hints to build a more modular version of the storage type system), but on the other hand, choosing the right strategy may require the use of an external reputation service. In contrast to PeerTrust, PeerAccess is purely a theoretical system, which means that there is no working implementation available.

The well-known *eXtensible Access Control Markup Language* (XACML) [77] supports distributed policies and also provides a profile for role based access control (RBAC). However, in XACML, it is the responsibility of the *Policy Decision Point* (PDP) – an entity handling access requests – to know where to look for the missing attribute values in the request.

4.7 Conclusions

In this chapter we introduce Core TuLiP, a trust management language based on Logic Programming. Core TuLiP forms the basis for the TuLiP trust management language which includes user-defined constraints and shares most features of moded logic programs, including practical features, such as interfacing with other languages, debugging facilities, etc.

Core TuLiP has all the advantages of trust management languages: for instance, a statement may be issued by one authority and it may be stored by an authority different from the issuing one. To deal with the problem of finding a credential when it is needed for a proof, we define the notion of a traceable credential and present a Lookup and Inference AlgoRithm (LIAR), which we show to be sound and complete w.r.t. the standard declarative semantics. We also compare Core TuLiP with RT_0 and show that each RT_0 credential and goal translates into Core TuLiP equivalent (with the small, but as we have argued permissible, exception of the goals of Sort 3).

The most important theoretical contribution of this chapter is that we show that it is possible to define a true TM language without leaving the well-established LP formalism. The practical relevance lies in the greater flexibility, extensibility and accessibility that an LP language enjoys with respect to – for instance – RT. As we have discussed, to accommodate various needs, the language RT_0 has developed a relatively large number of extensions, which make the language less flexible and harder to learn and understand. We thought that this was

the price we had to pay to have a true TM language, but Core TuLiP shows that this can be done otherwise.

CHAPTER 5

Trust Management in P2P systems using Standard TuLiP

In Chapter 4 we introduce Core TuLiP - the theoretical foundations of TuLiP - a Trust management Language based on Logic Programming.

Having a good theory, however, does not yet guarantee that this theory can be successfully implemented and used in practice. Therefore, in this chapter we investigate practical issues that must be solved when deploying Core TuLiP. We present a concrete version of Core TuLiP which is a trust management system consisting of a trust management language based on Core TuLiP, the description of necessary system components and the specification for the required underlying infrastructure. We call our system *Standard TuLiP*.

The contents of this chapter was first published as M. R. Czenko, J. M. Doumen, and S. Etalle. *Trust Management in P2P Systems Using Standard TuLiP*. In *Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security*, Trondheim, Norway, volume 263/2008 of IFIP International Federation for Information Processing, pages 1-16, Boston, May 2008. Springer.

5.1 Introduction

Standard TuLiP is based on the theoretical foundation laid by Core TuLiP (Chapter 4) i.e. on the same concept of a credential storage system and on a similar decision algorithm. But while Core TuLiP is more or less a theoretical exercise based on a restricted syntax, Standard TuLiP is a full fledged trust management system with not only a more flexible syntax, but with the support of a whole distributed infrastructure, with APIs for the specification, the validation and the storage of the credentials, APIs for interrogating the decision procedure and a number of changes w.r.t. Core TuLiP which makes Standard TuLiP amenable for a practical deployment (to mention one, the choice of including the mode in the credential specification, which allows to reduce the workload of the lookup algorithm). Having acquired the practical experience from the deployment of Standard TuLiP, in Chapter 6 we present the full version of TuLiP where we extend the expressive power of the language further and where we formalise the crucial implementation related concepts introduced informally in this chapter.

The chapter is structured as follows. Section 5.2 introduces the XML syntax of Standard TuLiP credentials and policies and shows how they are represented in the logic programming form. In Section 5.3 we present the architecture of Standard TuLiP. We introduce basic components, show their functionality and also state how they communicate with each other. In particular we show how credentials are stored and how we find them. Then, in Sect. 5.4 we show the system from the user perspective: we answer questions like how to write credentials, send queries, and we also discuss the problem of credential and user identifier revocation. In Sect. 5.5 we give additional insight into our concrete implementation of a distributed P2P content sharing system and which design choices we had to consider during our work. We finish the chapter with Related Work in Sect. 5.6 and Conclusions and Future Work in Sect. 5.7.

5.2 Policies

Standard TuLiP is a credential-based, role-based trust management system. Recall that, informally speaking, a credential is a signed statement determining which role can be assigned to an entity. A role can then be further associated with permissions, capabilities, or actions to be performed. For example, the University of Twente may issue a credential saying that *Alice* is a student of the University of Twente, which directly or indirectly may give Alice a certain set of permissions (like buying a book in an online store at a discount price). Here, the University of Twente is the *issuer* of the credential, Alice is the credential *subject*, and student is the *role name*. In Standard TuLiP, a credential is always signed by the credential issuer, as it is the issuer who has the authority of associating certain rights with the subject. A Standard TuLiP credential can also contain additional information about the issuer and/or the subject. For instance, a student usually has a student number, she belongs to a certain department, etc. This information is stored in the *properties* section of a credential. Standard TuLiP uses XML [97] as the language for the credential representation. The use of XML is convenient for several reasons. First, XML is a widely accepted medium for electronic data exchange and is widely supported by many commercial and free tools. Second, the use of XML namespaces [98] can help in avoiding name conflicts in the markup and facilitates the definition of

Listing 5.1: A basic Standard TuLiP XML credential for: *student_oi(ut-pub-key,alice-pub-key,[studentid:0176453,department:ewi,study:cs])*

```
<?xml version="1.0" encoding="UTF-8"?>
<credential xmlns="urn:ewi:namespaces:tulip"
notBefore="2007-02-12T20:00:00" notAfter="2008-02-12T20:00:00">
    <permission>
        <rolename>student</rolename>
        <mode>oi</mode>
        <issuer><entityID>ut-pub-key</entityID></issuer>
        <subject><entityID>alice-pub-key</entityID></subject>
        <properties>
            <studentid>0176453</studentid>
            <department>ewi</department>
            <study>cs</study>
        </properties>
    </permission>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
            ...
            <Reference URI="">
                ...
                <DigestValue>5WlwStu5ouu94nb5rwQ6BhFOPWc=</DigestValue>
            </Reference>
        <SignedInfo>
        <SignatureValue>signature-value</SignatureValue>
    </Signature>
</credential>
```

common vocabulary (recall that we have a separate namespace for user identifiers and role names).

We distinguish two types of credential: the basic credential, and the conditional credential. The basic credential is just a direct role assignment (e.g. "Alice is a student"), while the conditional credential can express role assignments under some constraints.

Basic credentials. Figure 5.1 shows the XML encoding of a basic Standard TuLiP credential (in the caption of Fig. 5.1, the second line shows the logic programming representation of the credential introduced later in this section - see Definition 5.2.1). The top level XML element is *credential*. The *credential* XML element, in turn, contains one *permission* XML element, which consists of the following XML elements: *role name*, *mode*, *issuer*, *subject*, and optionally *properties*. As in Core TuLiP (see Chapter 4), the mode indicates the storage location of the credential. Here we use a notation which is easier to parse (the mode of an argument is simply given by the index of the argument position), and we have $i = In$, $o = Out$, and so $oi = (Out, In)$. The *issuer* element consists of a single *entityID* element which contains a public identifier of the credential issuer. Similarly, the *subject* element contains the *entityID* element containing the public identifier of the subject. The

optional *properties* XML element can include arbitrary XML content describing additional properties of the issuer and/or the subject. Every credential includes the time period in which it is considered valid - this is expressed by the XML attributes *notBefore* and *notAfter* that are inside the top-level *credential* XML element. The credential is signed by the private key belonging to the credential issuer. Standard TuLiP uses public (RSA) keys as public identifiers. By doing this, every credential can be immediately validated without the need for an external PKI infrastructure (see also Sect. 5.3 for more information on the requirements Standard TuLiP have on the underlying infrastructure). This is because the public key needed to verify the signature on the credential is already present in the *issuer* element embedded in the *permission* element of the credential. The signature is contained in the *Signature* XML element. We use the enveloped XML signature format [96] (more precisely, a digest value is computed over the top-level element, which is then included in the DigestValue element of the SignedInfo element of the signature, and then the signature is made of the SignedInfo element and included in the SignatureValue element). The *xmlns* attribute in the top-level *credential* XML element contains the namespace identifier *urn:ewi:namespaces:tulip*. This is required for every valid Standard TuLiP credential. By using namespaces, Standard TuLiP credentials can be distinguished from other credential formats and this even allows for different credential formats to be mixed in a single XML document.

Conditional credentials. Basic credentials are insufficient to model more sophisticated statements, for instance including delegation. Consider an online store which gives a discount to a student of the University of Twente. Instead of giving each student a basic credential granting the discount, it is more efficient to associate the discount role to everyone who has the student role at the University of Twente. In order to express this kind of statement we need a *conditional* credential.

Listing 5.2: A conditional Standard TuLiP XML credential for:
discount_ii(*eStore-pub-key*,X,Y) ⟵ *student_oi*(*ut-pub-key*,X,Y)

```
<?xml version="1.0" encoding="UTF-8"?>
<credential xmlns="urn:ut:ewi:namespaces:tulip"
 notBefore="2007-02-12T20:00:00" notAfter="2008-02-12T20:00:00">
    <permission>
        <rolename>discount</rolename>
        <mode>ii</mode>
        <issuer><entityID>eStore-pub-key</entityID></issuer>
        <subject><var>X</var></subject>
        <properties><var>Y</var></properties>
    </permission>
    <provided>
        <condition>
            <rolename>student</rolename>
            <mode>oi</mode>
            <issuer><entityID>ut-pub-key</entityID></issuer>
            <subject><var>X</var></subject>
            <properties><var>Y</var></properties>
        </condition>
    </provided>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
```

```
            ...
22      </Signature>
    </credential>
```

A conditional credential consists of one *permission* XML element followed by one *provided* XML element (Fig. 5.2). The *permission* element has the same structure as the *permission* element in the basic credential. The *provided* XML element includes one or more *condition* elements, each of which specifies an additional condition that must be satisfied before the specified role name can be associated with the credential subject. A *condition* XML element is similar to the *permission* XML element in that it also contains the XML elements *role name*, *mode*, *issuer*, *subject* and optional *properties*. In conditional credential a variable can be used to make a logical link between two or more XML elements occurring in the credential. The only exception is the issuer of a credential (the *issuer* element inside the *permission* element) that must not contain a variable as the issuer of a credential must always be known (otherwise it would not be possible to verify the signature on a credential).

Figure 5.2 shows the credential expressing the following statement: *eStore* associates the discount role name to all students of the University of Twente (*ut*) (for conciseness we omit the contents of the signature XML element). The *subject* XML element inside the *permission* XML element contains variable X, which links this element with the *subject* XML element inside the *condition* XML element in the *provided* part of the credential. This means that *eStore* assigns the discount role name to every entity X for which the *condition* is satisfied, i.e. to every student of the University of Twente. Similarly, the *properties* element inside the *permission* element contains variable Y which is then used in the *properties* element inside the *condition* element in the *provided* part of the credential. This means that when discount is granted, all the properties of a student are included in the discount atom (and then used e.g. for logging). To prove that Alice is eligible for a discount at *eStore* both the credential presented in Fig. 5.1 and the credential presented in Fig. 5.2 are needed.

The *provided* part of a conditional credential can also contain a constraint which in turn can refer to a built-in function (e.g. in order to manipulate values taken by the variables). The presence of variables also allows us to interface with external components (e.g. arithmetic solvers, constraint solvers, programs written in other languages). The only requirement is that the variables used should respect the input-output flow dictated by the modes in the credential (the modes are introduced in Chapter 4). In Chapter 6 we show how an external evaluation algorithm is used to evaluate user-defined constraints. Interfacing with other external components is part of the future work.

A Standard TuLiP security *policy* is defined by a set of credentials, i.e. a set of XML documents.

Queries. When *eStore* wants to check if *Alice* is a student of the University of Twente it sends a *query* to the University of Twente. In Standard TuLiP, a query is also encoded as an XML document. Figure 5.3 shows an example Standard TuLiP query, in which the issuer of the query (a user using public key *jeroen-pub-key*) tests if *Alice* can have a *discount* at *eStore* and if so retrieves the associated properties (through variable X). The structure of a Standard TuLiP query is similar to that of the *provided* part of a Standard TuLiP XML credential. The top-level element is the *query* XML element. It consists of a one or more *condition* XML elements each of which contains XML elements *role name*, *mode*, *issuer*,

subject, and optionally the *properties*.

Listing 5.3: A Standard TuLiP XML query *discount_ii*(*eStore-pub-key*,*alice-pub-key*,*X*).

```
<?xml version="1.0" encoding="UTF-8"?>
<query xmlns="urn:ut:ewi:namespaces:tulip"
 ID="tulip-ut-ewi-2f72ca8f-25bc-4d50-b1eb-3861e42f1562"
 IssueInstant="2007-06-29T04:04:38">
    <issuer>jeroen-pub-key</issuer>
    <condition>
        <rolename>discount</rolename>
        <mode>ii</mode>
        <issuer><entityID>eStore-pub-key</entityID></issuer>
        <subject><entityID>alice-pub-key</entityID></subject>
        <properties><var>X</var></properties>
    </condition>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        ...
    </Signature>
</query>
```

Besides the query conditions, every query reports the public identifier of the entity making the query. Each Standard TuLiP query also contains a unique *ID* attribute and an *IssueInstant* attribute inside the top-level query XML element. The *ID* attribute allows the system to check whether the received response corresponds to the earlier issued query. The *IssueInstant* attribute carries the time and date of the request which allows the responding entity to filter out erroneous requests (like the ones with the time in the future), or to check whether the time matches the validity of a credential used in answering the query. Like a credential, a query is always signed by the query issuer.

A query is always about a specific set of permissions. However, there are different types of query. In Chapter 2 we distinguish three different sorts of query in RT_0 and Chapter 4 we show how RT_0 queries translate to Core TuLiP queries. We have therefore (we use the RT_0 notation of a *role* from Chapter 2):

Sort 1: "Given two entities a and b, and a role name r, check if b is a member of role $a.r$." This query translates to r_*mode*(a, b, X) in Standard TuLiP where *mode* $\in \{ii, io, oi\}$ and X will be instantiated with the properties assigned to the subject.

Sort 2: "Given an entity a and a role name r, list all members of role $a.r$." This query translates to r_io(a, X, Y) in Standard TuLiP where, for each entity b being a member of role $a.r$, X will be instantiated with b and Y with the properties of b.

Sort 3: "Given an entity b find all roles $a.r$ b is a member of." This query has no equivalent in Standard TuLiP (see the discussion below).

For the reasons given in Chapter 4, in Standard TuLiP we support only first two sorts. For instance "Is *alice* a student of the University of Twente?" and "Give me all the students

of the University of Twente" are valid queries of Sort 1 and Sort 2 respectively. As in Core TuLiP, in Standard TuLiP the policy writer can restrict the type of a query one can ask by using modes (see Chapter 4 for the detailed discussion).

Semantics. The semantics of Standard TuLiP follows the semantics of Core TuLiP in that each Standard TuLiP credential is given a logic programming representation. In this representation every credential is represented by a definite clause containing one or more the so called *credential atoms* and/or *built-in constraints*. Unlike Core TuLiP, where each credential atom has two arguments, in Standard TuLiP, a credential atom has three arguments to account for the properties.

Definition 5.2.1 (credential atoms,credentials,queries) *A* credential atom *is a predicate of the form:*

$$\textit{rolename\_mode}(\textit{issuer}, \textit{subject}, \textit{properties}).$$

A credential *is a definite clause of the form* $P \longleftarrow C_1, \ldots, C_n$*, where* P *is a credential atom, and* $C_1, \ldots, C_n$ *are credential atoms or built-in constraints. The credential atom* P *in the head of the clause corresponds to the permission XML element and every credential atom or built-in constraint* C_i *in the body of the clause corresponds to a condition in the provided part of the corresponding XML credential encoding. A query is represented by a sequence of credential atoms and/or built-in constraints* $C_1, \ldots, C_n$*, where each* C_i *corresponds to a query condition.*

The var XML element maps to a logical variable. We present the actual mapping between the *properties* XML element and the corresponding logic programming term in Chapter 6.

Finally, a Standard TuLiP policy is represented by a logic program.

Example 5.1 The credentials presented in Figs. 5.1 and 5.2 and query presented in Fig. 5.3 have the following logic programming representation:

(5.1) *student_oi*(*ut-pub-key*, *alice-pub-key*, *[studentid:0176453,department:ewi,study:cs]*).

(5.2) *discount_ii*(*eStore-pub-key*, X, Y) $\longleftarrow$ *student_oi*(*ut-pub-key*, X, Y).

(5.3) *discount_ii*(*eStore-pub-key*, *alice-pub-key*, X).

Here, *[studentid:0176453,department:ewi,study:cs]* in credential (5.1) is the Prolog term corresponding the "properties" XML element.

Recall that given a role name r, the set of all credentials having p as a role name of the credential atom occurring in the head is called the *definition* of r. Every credential from the definition of r is called a defining credential of r.

For sake of conciseness, in the remainder of this chapter we will refer to the logic programming representation of a credential instead of using the original XML encoding. For the same reason, we will sometimes write a credential atom without the last *properties* argument if it is clear from the context what is meant.

So far, we have introduced the credentials, the queries and the semantics. Now we need a system which, given a query, finds the credentials required for the query evaluation and returns the result of this evaluation to the issuer of the query.

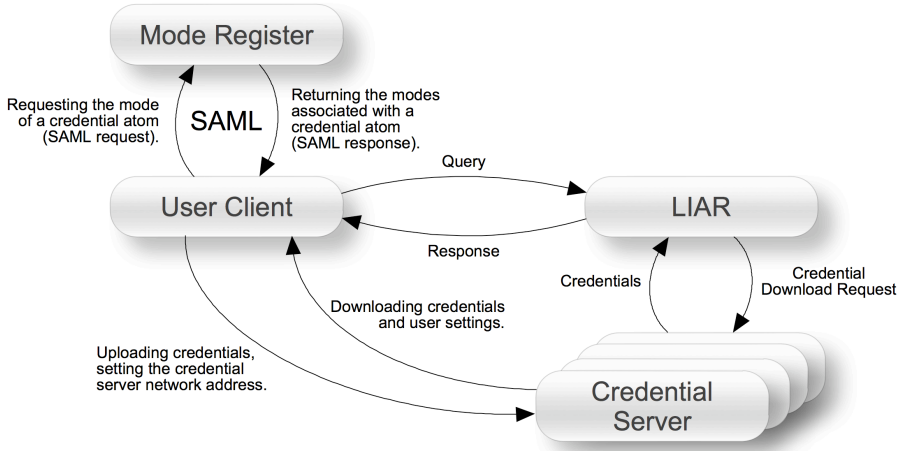


Fig. 5.1: Components of Standard TuLiP

5.3 System Architecture

In this section we describe the architecture of Standard TuLiP. First, we present the system components. Then we show how the system components interoperate and we give a concrete example demonstrating this. Finally, we present the requirements Standard TuLiP has on the underlying infrastructure. In particular, we discuss how a public identifier can be mapped to a physical network address.

5.3.1 System Components.

In Standard TuLiP we identify the following components: (a) the LIAR engine, (b) the credential server (c) the User Client application, and (d) the mode register (see Fig. 5.1).

By default, every system user runs an instance of the LIAR engine, but other approaches are also possible. For instance, there can be a preselected set of nodes running LIAR, or there can even be only one instance of LIAR serving a whole community. Because LIAR relies on unification and – to some extent – on backtracking, the most natural way of implementing the LIAR algorithm is to reuse an existing Prolog engine. Using Prolog makes deployment of LIAR easier simultaneously allowing us to preserve the original logic programming formalism in the "reasoning" part of the system. We implement LIAR using YAP Prolog [68], which is a freely available Prolog implementation with several optimisations for better performance. The external network interface is written in Python. LIAR operates as an HTTP server when answering the queries and as a client when fetching credentials from *credential servers*. We give a more detailed functional description of LIAR later in this section.

In Standard TuLiP every user is associated with one credential server. It is possible, however, that one credential server serves many users. The credential server responds to a credential request coming from a LIAR engine and returns credentials satisfying this request. The credential server is implemented as a *simple* HTTP server (written in Python) and is internally connected to a *credential store*, which stores all user's credentials.

The *User Client* is a GUI application (written in Flash and Python) and provides a user-friendly interface to other Standard TuLiP system components. In particular, the User Client is used for: generating the user private-public key pair, setting up and maintaining the location of the credential server corresponding to the user, importing user credentials, and querying the Standard TuLiP system. Optionally, an additional application in the form of a plugin can be provided. For instance, one could provide a plugin having a graphical credential editor functionality. The User Client application itself does not allow the user to perform any action on a remote resource. Its main purpose is to let the user query the system.

Another important component of Standard TuLiP is the *mode register*. The mode register stores the modes associated with the role names used in the system. Currently, the mode register is a centralised service operated by the University of Twente. For our initial implementation, the centralised version of the mode register is sufficient, especially when taking into account that the mode information is included in each Standard TuLiP credential and query. It means that mode register is needed mostly when writing credentials in order to check the consistency of the mode assignment (see also Sect. 5.4). During normal operation, the mode register is rarely needed. This can happen if the mode register becomes inconsistent (for instance because of a non-authorised use or a system failure). The mode register may also be helpful in finding the storage inconsistencies - when we cannot find a credential at the expected location and the mode information encoded in the credential agrees with that from the mode register, we know that the problem is in the inconsistent storage and we know which entity to query. Because storage inconsistencies are not expected to happen often, making the mode register a centralised service has minimal impact on the performance and reliability of the whole system. A more hierarchical and distributed approach is also possible to distribute the ownership of the service and increase reliability and comfort of use (the service must be operational, otherwise issuing a new credential may lead to inconsistency in the credential storage). The mode register is implemented as an HTTP server with user-friendly web-interface and supports the Security Assertion Markup Language (SAML) protocol [78].

SAML is an XML-based framework for communicating user authentication, entitlement, and attribute information. In Standard TuLiP, each role name is represented by a SAML *assertion* [76]. In order to retrieve the modes associated with a given role name we use a SAML *attribute query* in which as we pass a role name as the ID of a subject and *urn:ut:ewi:names:tulip:resource:mode* as the id of an attribute to be queried. In other words, a role name is seen as a subject and the associated modes as the values of the "mode" attribute of this role name. Listing 5.4 shows an example SAML attribute query (here we show a real request with the real public key and the signature).

Listing 5.4: SAML Attribute Query for the "accredited" role name.

```
<?xml version="1.0" encoding="UTF-8"?>
<AttributeQuery xmlns="urn:oasis:names:tc:SAML:2.0:protocol"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:SAML:2.0:protocol
                      http://docs.oasis-open.org/security/saml/v2.0/saml-schema-protocol-2.0.xsd"
  ID="tulip-ut-ewi-0eb10a41-2389-4d63-9677-afbab460142e" Version="2.0"
  IssueInstant="2008-08-17T21:13:42" Destination="http://tulip-ut.nl/getMode.php">
    <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsgV0HIxl2qoyBrQJMK07
IuBUBt9NNqG2158SDphJzuX/ETGrv2F/3mZxPqPL0JLhg55tPZmQoU1EDJ1cz0R9
```

```
d9Z6JiycpKFXYLyDHv1t/Ewahvj2RSD2mBBNDkFw+r+Gn1ocoCRl/PmpSE0U4qet
CekRoWdQtzqelmCqHGsY22/V1G9UrnAJIN2YEWjFrKXj+s9bLKMSHuRESNCSxdVL
jM5wGo16maYGY01OUiuhQIX06waW6xYIRJ7jKCjEDEC/YmfNEZc1rctVpWLwgLS2
ZcSrK9xIHQZfsxPQYqFji8TrOOLuyjXHQJDj0ebD3xt3XPWzXJO063S+ycFgAhmd
SQIDAQAB
</Issuer>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml−exc−c14n#"/>
      <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa−sha1"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped−signature"/>
          <Transform Algorithm="http://www.w3.org/2001/10/xml−exc−c14n#"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue>B2OVfx53JVy7e1LcOuT8ufrFvrM=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>
aZWZGSCib/WWgYYKyEHA3kuFuwUZerTNSP6DeTw3Yef8v2PD9hhmqxT5+VrTM08f
/KI/OnoabE7YG/M8VBJcUVjOY1l7ujyy+7MSqf72pYInpBlnFQ8Zw8fPZxk4XRND
bUm5la2YwsARRCZYv5UlGzzj8d83qXGQgTZKSAqebMgenGCNHFFXU9yRU/GnxJ6j
x9k2AO1nS4yx7EkdCLOru4uAlZyqc2F1j3FQvua3Z0oAHAwpjv6JoDromlibogmg
5AFjvBF1N5d60TBbTa1NtQkiRvVp4X3sjFWk7sM5eyI/8k/pEcn/VVuEqBHFchET
NkQxRbed5o7xUsZ8SGutRA==</SignatureValue>
    </Signature>
    <Subject xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
        <NameID>accredited</NameID>
    </Subject>
    <Attribute xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
     Name="urn:ut:ewi:names:tulip:resource:mode"/>
</AttributeQuery>
```

The mode register responds to an SAML attribute query by returning an SAML *response* [76] containing one or more assertions. Each returned assertion holds one role name and the modes associated with this role name. Listing 5.5 shows the SAML response corresponding to the SAML attribute query shown in Listing 5.4.

Listing 5.5: SAML Assertion corresponding to the query in Listing 5.4

```
<?xml version="1.0" encoding="UTF−8"?>
<Assertion xmlns="urn:oasis:names:tc:SAML:2.0:assertion"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
 xsi:schemaLocation="urn:oasis:names:tc:SAML:2.0:assertion
              http://docs.oasis−open.org/security/saml/v2.0/saml−schema−assertion−2.0.xsd"
 Version="2.0" ID="tulip−ut−ewi−b35dc804−4b99−a339−e95b−8181ee863830"
 IssueInstant="2008−08−17T21:13:45">
    <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAthUolTlzjcUy/7cJ3eVf
KVH9fved7AROsQRYM6+gJJgigBL/wsex5gr2iCuh/c3PHnAejRZ13eIMqY3Gk7/y
```

```
i8GkTgByxBZALrJWHkHqG96ZaobG5qcGLaaRGzkflgs1EqcnMrsb6TOVfHEiJlhm
tZhQsONjDWYZdMRHaalcp1roPxoX+3Ftit4PCj/LNhnsk0aRX5pmHem2a66EROkT
sLm31GRazC6rOmvCwD+TTYiUqEbucyJDLas4OQZfAglD8vLsUR2V5IXelVkLBdfN
e1u1EjCk4lHK0QGGNtEGv6YuI9I73exqNpK4F0kd35YPL53SM63D+ab61d74ZGjY
SwIDAQAB
    </Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            <ds:SignatureMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
            <ds:Reference>
                <ds:Transforms>
                  <ds:Transform
                    Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
                  <ds:Transform
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
                </ds:Transforms>
                <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                <ds:DigestValue>OC/a584DnEAYCGIKNjyD86Ff7oU=</ds:DigestValue>
            </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>
Cs4zBEcQDJpzirqC5c1g7uhil03w/J9DvLQROVNZzHU5JmZA+Y+6hQ/we2EjTLNs
Kqu90Nr6Bqit+HsrLDZbjIHipyEx+rRCzceENZr2fu5nl59hw/qWHOYLYmAN9LBmnVj
xwSQdZh16k071Vz8vnxIzFPKdZNBJNlkTOX5dCnX9fFhFufSOjymCnpnRigLwe7tzE
QQASLf3eg94H5vwbkzpqRF0d6yARdaN0HvBGoWkigLkwXnYd92DeKvuJDo1ZfjCH
5r8+zt4g+3vUEYcsTF6tGK6k8hLY3iLH4gr7hsb6bFQJ6Pz+kPI7GaMiIRzK9PyVc5Ku
AVyRhKRgrg==
        </ds:SignatureValue>
    </ds:Signature>
    <Subject>
        <NameID>accredited</NameID>
    </Subject>
    <AttributeStatement>
        <Attribute Name="urn:ut:ewi:names:tulip:resource:mode">
        <AttributeValue>II</AttributeValue>
        <AttributeValue>IO</AttributeValue>
        <AttributeValue>OI</AttributeValue>
        </Attribute>
    </AttributeStatement>
</Assertion>
```

Currently, the mode register uses the HTTP POST binding to carry the SAML attribute queries and responses [79]. The mode register uses version 2.0 of the SAML standard.

We expect that an appropriate implementation-level mechanism can be used to guarantee the confidentiality, integrity, and non-repudiation of the messages being exchange between the system components. For instance in our proof of concept implementation we support non-repudiation by using XML signatures while we do not force messages to be encrypted. We

leave it to the implemention to decide a concrete security technology to be used in different parts of the system.

5.3.2 LIAR.

The basic functionality of LIAR is to (1) accept a query, (2) find the (possibly negative) proof for the query (3) when building the proof fetch the required credentials, and (4) return a response to the user.

When LIAR receives a query (recall that each Standard TuLiP query is an XML document) from the User Client it first checks the signature on the query and then LIAR starts the evaluation process. Every time an additional credential is needed, LIAR fetches the missing credential from the location indicated by the mode of the credential atom being evaluated. By embedding the mode information in each credential and in each query, the mode register does not have to be contacted in order to determine the storage location of the credentials defining a given role name. The credentials are fetched from the corresponding credential server by sending a so called *credential request*. A credential request is an XML document specifying which credentials should be fetched. Listing 5.6 shows an example of a Standard TuLiP credential request in which the issuer of the request asks for all the credentials for which the permission element matches (by logical unification) the permission element in the request.

Listing 5.6: Example Standard TuLiP Credential Request for each credential whose head unifies with *discount_ii*(*eStore-pub-key*,*alice-pub-key*,X).

```
<?xml version="1.0" encoding="UTF-8"?>
<credentialRequest xmlns="urn:ut:ewi:namespaces:tulip"
 ID="tulip-ut-ewi-1bcfd26b-0c9f-477f-adc9-aa42f9b30aa1"
 IssueInstant="2007-06-29T04:04:54">
    <issuer>jeroen-pub-key</issuer>
    <permission>
        <rolename>discount</rolename>
        <mode>ii</mode>
        <issuer><entityID>eStore-pub-key</entityID></issuer>
        <subject><entityID>alice-pub-key</entityID></subject>
        <properties>
            <var>X</var>
        </properties>
    </permission>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        ...
    </Signature>
</credentialRequest>
```

Besides the *permission* element that is matched against the permission element of one or more credentials stored on the credential server, the Standard TuLiP credential request also specifies the issuer of the request. In Listing 5.6 the issuer of the request is a user using the public key *jeroen-pub-key*. If any credential stored at the credential server matches the permission element of the credential request, such a credential is returned to the issuer of

the request. LIAR validates each received credential by checking the signature and validity interval on the credential.

After evaluating a query, LIAR sends to the User Client the so called Standard TuLiP *response* (XML) document containing all the answers, i.e. for each query condition it returns one or more instances satisfying this query condition. The top-level element of the Standard TuLiP response is the *response* XML element. Besides the unique *ID* and *IssueInstant* XML attribute it also contains *InResponseTo* XML attribute containing the value of the *ID* XML attribute from the corresponding query. Listing 5.7 shows an example of the Standard TuLiP response document.

Listing 5.7: Example Standard TuLiP Response for query *discount_ii*(*eStore-pub-key*,*alice-pub-key*,*X*) containing one answer *discount_ii*(*eStore-pub-key*,*alice-pub-key*,*[studentid:0176453,department:ewi,study:cs]*).

```
<?xml version="1.0" encoding="UTF-8"?>
<response ID="tulip-ut-ewi-971a79f2-9a50-4838-8d0a-2f6657613dba"
 InResponseTo="tulip-ut-ewi-2f72ca8f-25bc-4d50-b1eb-3861e42f1562"
 IssueInstant="2007-06-29T04:04:55">
    <issuer>jeroen-pub-key</issuer>
    <permission>
        <rolename>discount</rolename>
        <mode>ii</mode>
        <issuer><entityID>eStore-pub-key</entityID></issuer>
        <subject><entityID>alice-pub-key</entityID></subject>
        <properties>
            <studentid>0176453</studentid>
            <department>ewi</department>
            <study>cs</study>
        </properties>
    </permission>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        ...
    </Signature>
</response>
```

A Standard TuLiP response contains one permission element for each successful instance of a query. If a query fails, the response does not contain any permission element. The issuer of a Standard TuLiP response is the entity which runs an instance of the LIAR algorithm which received the initial query.

The following example demonstrates the system behaviour in the response to a concrete query.

Example 5.2 Assume we have the following set of credentials (we use the logic programming notation as shown in Sect. 5.2):

(1) $\mathit{discount\_ii}(\mathit{eStore\text{-}pub\text{-}key}, X) \longleftarrow \mathit{accredited\_io}(\mathit{accBoard\text{-}pub\text{-}key}, Y), \mathit{student\_oi}(Y, X).$

(2) $\mathit{accredited\_io}(\mathit{accBoard\text{-}pub\text{-}key}, \mathit{ut\text{-}pub\text{-}key}).$

(3) $\mathit{student\_oi}(\mathit{ut\text{-}pub\text{-}key}, \mathit{alice\text{-}pub\text{-}key}).$

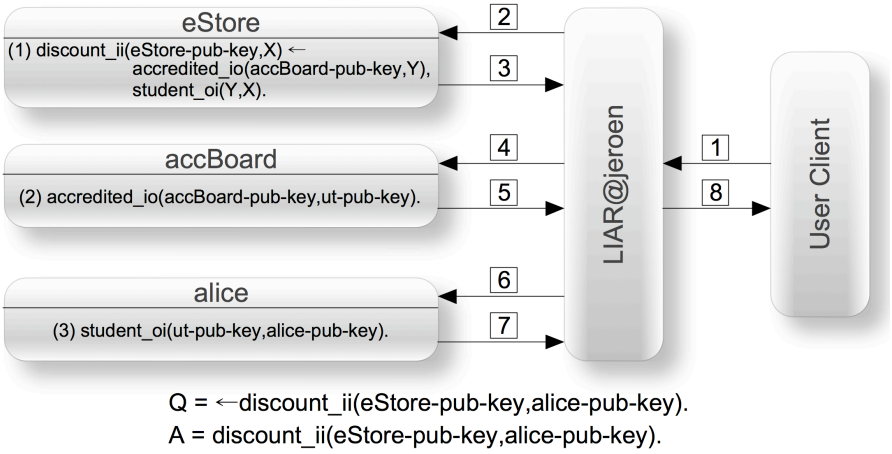


Fig. 5.2: Credential Discovery with LIAR

Figure 5.2 presents the steps performed by LIAR during evaluation of the query $\longleftarrow$ *discount_ii*(*eStore-pub-key*, *alice-pub-key*). In Fig. 5.2 each rounded rectangle represents a credential server associated with the entity of which name is printed at the top of the rectangle, each arrow represents a message, and the number above each arrow represents the message order. Below, we show the execution of the algorithm for the above mentioned query.

We assume that the instance of the LIAR algorithm is run by a user Jeroen with the public identifier *jeroen-pub-key*. In message $\boxed{1}$ LIAR receives the query in which the query issuer (Jeroen) asks whether the user with public id *alice-pub-key* has a discount at the internet store identified by *eStore-pub-key*. The query is signed by the query issuer. Before evaluating the query, LIAR checks the signature on the query, then it checks the mode of the atom *discount_ii*(*eStore-pub-key*, *alice-pub-key*). Because the mode is *ii* $(=(In, In))$, LIAR knows that it should try to fetch credentials matching this query from the issuer of *discount_ii*(*eStore-pub-key*, *alice-pub-key*) which is *eStore*. This is done in messages $\boxed{2}$ and $\boxed{3}$. After receiving the matching credentials, LIAR validates each of them, which means that for each fetched credential LIAR checks the signature and the validity interval. Next, every successfully validated credential is instantiated by unifying the head of this credential with the query atom. In our case only one credential is fetched (credential (1)) and the resulting instance is:

$$\textit{discount\_ii}(\textit{eStore-pub-key}, \textit{alice-pub-key}) \longleftarrow \textit{accredited\_io}(\textit{accBoard-pub-key}, Y), \textit{student\_oi}(Y, \textit{alice-pub-key}).$$

We see that in order to prove the initial query, now LIAR has to evaluate the following query:

$$\longleftarrow \textit{accredited\_io}(\textit{accBoard-pub-key}, Y), \textit{student\_oi}(Y, \textit{alice-pub-key}).$$

In evaluating this (sub) query, LIAR first checks the mode associated with atom

accredited_io(*accBoard-pub-key*, Y). Because the mode is *io* ($=(In, Out)$), the defining credentials (if any) should be stored by *accBoard*. The *accBoard* is queried in message $\boxed{4}$, resulting in fetching credential (2) in message $\boxed{5}$. The credential is then validated, and then the instance of this credential - *accredited_io*(*accBoard-pub-key*, *ut-pub-key*) - is used. As the body of credential (2) is empty, the main query reduces to $\longleftarrow$ *student_oi*(*ut-pub-key*, *alice-pub-key*). The mode of *student_oi*(*ut-pub-key*, *alice-pub-key*) is *oi* ($=(Out, In)$) which means that related credentials should be stored by the subject: *alice* in this case. Alice is contacted by LIAR with message $\boxed{6}$, and asked for all credentials moded *oi* ($=(Out, In)$) she stores (see Chapter 4 for the explanation why LIAR fetches all the credentials moded (Out, In) and not only credentials having role name *student* and mode (Out, In)). In the response, in message $\boxed{7}$, credential (3) is returned and then validated. This credential unifies with *student_oi*(*ut-pub-key*, *alice-pub-key*) and at this point the original query has been evaluated successfully. The information about successful evaluation, containing only one permission element corresponding to the *discount_ii*(*eStore-pub-key*, *alice-pub-key*)) credential atom, is sent to the User Client in message $\boxed{8}$.

5.3.3 Public Identifiers.

Recall that Standard TuLiP uses a public key as a public identifier of a user. In Example 5.2 we have silently assumed that there exists a mapping from each public identifier to a concrete network addres. Indeed, Standard TuLiP requires an underlying service to map public identifiers to concrete network addresses.

Distributed Hash Tables (DHT) [85] represent a class of overlay P2P systems with key-based routing functionality. They provide a look up service similar to a hash table. In a DHT system, data are distributed across many *nodes*. Each node itself provides a hash table functionality: each block of data (a *value)* is identified by a *key*. One can store a new key and value pair or, knowing the key, one can read the associated value. The node storing the value corresponding to the given key can be usually identified by this key. For instance, in the Kademlia DHT system [70] each node is identified by a *nodeID* which is a 160-bit binary number. The key has the same format as the nodeID. A key in Kademlia DHT system (and also in most of other DHT proposals) is a hash (Kademlia uses 160-bit SHA1 digest) of the value (a data block to be stored under the given key). In a typical DHT lookup operation, the nodeID is determined first based on the key and then the same key is used to read the corresponding data. Therefore, the user may not be aware of the actual storage location for the data. Kademlia extends the Chord DHT system [94]. Other important DHT proposals are Pastry [88] and Tapestry [106]. Kademlia is also used in BitTorrent [24].

Standard TuLiP can take advantage of the lookup mechanism offered by DHTs. In this case the user does not have to provide a dedicated credential server but rather uses the storage mechanism provided by the concrete DHT implementation. When using a DHT, each user credential (normally stored at the credential server corresponding to this user) is stored at the DHT node corresponding to the user public identifier: the hash of the user public identifier becomes a key under which all user credentials and all other user related information would be stored (like the current IP address of the User Client application acting in the name the user). When DHT technology is used, the user cannot choose her own credential server and the security and reliability of the system is strongly influenced by the security and reliability

of the chosen DHT implementation. The Kademlia DHT system, for example, provides the services like data replication, resistance against denial of service attack and introduction of fake nodes to the system, and all this at a high performance O(log(n)) where n is the number of nodes.

5.4 Using Standard TuLiP

In using the Standard TuLiP trust management system we can distinguish the following actions that may be performed by the user: (1) issuing a credential, and (2) sending a query and receiving a response.Below we briefly summarise issues raised by these actions.

Writing Credentials. When issuing a credential one must be sure that any new credential is traceable. The User Client application helps in writing credentials by checking that each credential is traceable. Before accepting the credential, the User Client checks if *for every* mode value of the head of the credential there exists a permutation of the credential atoms occurring in the body of the credential and the corresponding mode values such that the credential is traceable. If this is not the case, the credential is refused. If for a mode value of the head there exists more than one valid mode assignment for the credential atoms in the body, the user will be allowed to choose a preferred one. For example, the user may have a preference to limit the number of subject traceable credentials in order to increase the reliability of the system.

The User Client application determines the mode of a credential atom by querying the mode register. The selected mode is then embedded into the credential so that the mode register does not have to be referred to during query evaluation later. Recall that the mode assigned determines the actual credential storage location. The User Client application automatically uploads the new credential to the suitable credential server (as given by the user record associated with the public id of the user).

When a user introduces a credential with a new role name, it has to be registered with the mode register. The mode register can be accessed through the TuLiP home-page, or by using a dedicated application. Each user can request the registration of additional role names and the corresponding modes by requesting it through the TuLiP web-site (*http://tulip-ut.nl*).

Writing Queries. Every Standard TuLiP query must be well-moded. Therefore, before sending a query, the User Client application checks for well-modedness. If some credential atom in the query has more than one mode value, it is possible that there will be more than one variant of mode assignment that makes the query well-moded. In such a case, the User Client lets the user select the preferred mode assignment (e.g. the one that is likely to yield the correct answer most efficiently). The User Client application sends the query to the LIAR engine associated with the user issuing the query and presents the received response.

5.5 Implementing TuLiP

We shown the general description of the system architecture and the required system components in Sect. 5.3. In this section we give the reader additional insight into our concrete

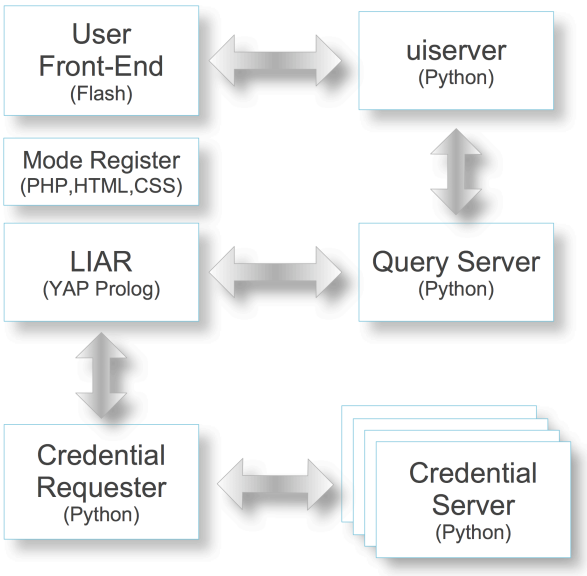


Fig. 5.3: The components in our proof of concept implementation of TuLiP

implementaion and the design choices we had to make.

The crucial part of our TuLiP system is the compliance checker LIAR (Fig. 5.3) We implement LIAR using Prolog. Prolog as a Logic Programming language is the most natural language to implement LIAR as Prolog provides the unification and backtracking, on which LIAR depends and which are not trivial to implement otherwise. As our Prolog engine we chose the YAP (Yet Another Prolog), which is an efficient Prolog implementation largely compatible with the ISO-Prolog standard.

In a real system LIAR needs to be able to receive the queries through the network interface. LIAR also needs to be able to fetch the credentials from the credential servers. Although YAP allows us to implement a client/server functionality directly in Prolog we chose a more scalable approach. We decided to keep "the Prolog part" of LIAR as general as possible. In particular we decided that LIAR should not directly deal with parsing of XML content (in the case when credentials are encoded using XML), checking signatures, time stamps and with other application dependent activities. For this reason, in our implementation, LIAR communicates with the outside world not directly but with the help of two other components which we call the *Query Server* and the *Credential Requester* (Fig. 5.3). The Query Server is a simple implementation of an HTTP server written in Python, which performs all the necessary validation of a query (like checking the signature and the time stamps - see also Chapter 5). Only if the validation succeeds, the Query Server forwards the query to LIAR. The Credential Requester handles the communication between LIAR and a *Credential Server*. When LIAR needs to fetch a credential it communicates with the Credential Requester and the Credential Requester creates the appropriate credential request (in the XML format), signs the request, and then sends the request to a credential server. The returned set of credentials is

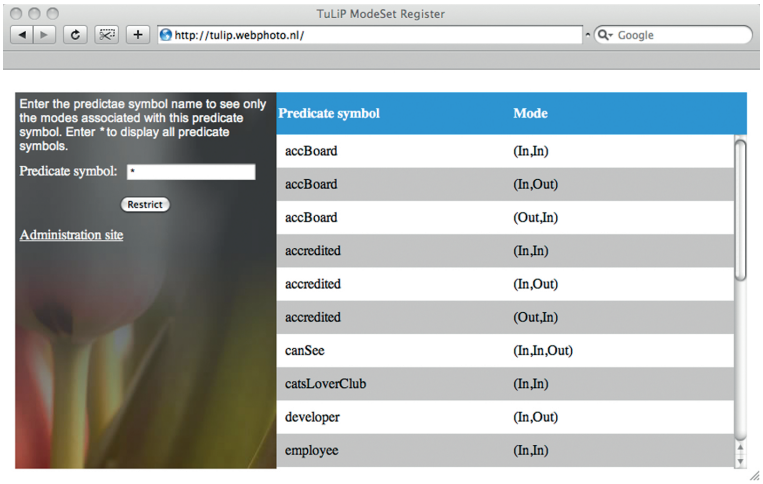


Fig. 5.4: The web interface to the Mode Register

validated and the resulting list of credentials are returned to LIAR. The Credential Server is also an HTTP server application written in Python.

One more component shown in Fig. 5.3 is the *User Interface Server* (uiserver). The role of this component is to separate the user interface (the front end) from the actual realisation of the underlying communication mechanism. This simplifies the creation of new front-ends as they can be simply "plugged" into the User Interface Server.

Although our demo implementation does not require the mode register to function, we built one as an example. The Mode Register can be accessed in two ways. The first way is through the web interface (Fig. 5.4). Each user can check which credential atoms are already registered and which mode is assigned to which credential atom. The web interface also allows the user to search for the specific credential atom and the modes in a convenient way by using the search functionality. A user having administration privileges (currently the TuLiP team at the UT) can add/remove credential atoms and change the modes through the administration site. Another way to access the mode register is through the use of the Security Assertion Markup Language (SAML) interface (see Chapter 5 for the description of SAML). The SAML interface is implemented in PHP. Here we would like to mention that it was a challenge to build a working implementation of the signing and validating operations in PHP. Finally, we provide a command-line application *Mode Checker* (written in Python), in which we demonstrate how to use the interface so that the user can build her own utilities.

As a standard, in our implementation messages are signed (we use the XML signature standard [96]) but not encrypted. The credentials are exchanged as base-64 encoded character strings (neither signed nor encrypted).

Based on our implementation, we provide a demo which demonstrates the application of TuLiP in the distributed content management. In our case the contents are the pictures. Each user stores a number of pictures. Pictures are divided into several user defined groups and the credentials stored by different users influence which user can access which group of pictures. Figure 5.5 shows the User Client (the front-end) application run by one of the users.

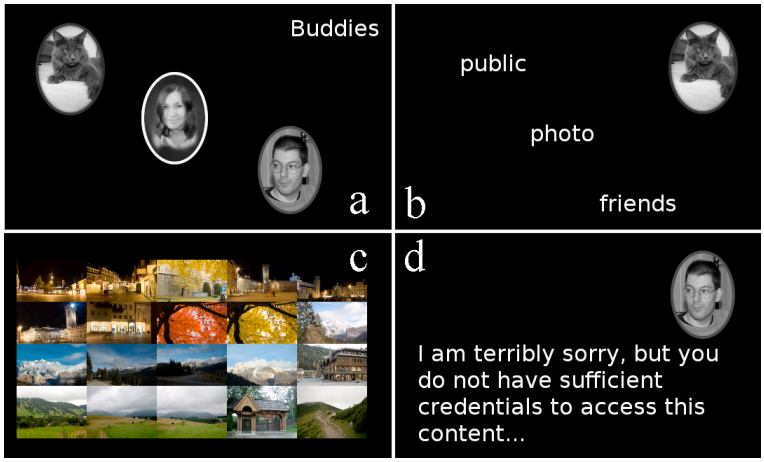


Fig. 5.5: The Front-End of our Content Management Demo System

Initially the user sees the list of known entities (we call them *buddies* in the figure). When the user selects a buddy, the front-end sends the request (through the User Interface Server - uiserver) to the appropriate LIAR engine and the query is evaluated. Based on the credentials discovered during the evaluation, the user gains access to several groups of pictures (Figs. 5.5 (b,d)). When the user does not have sufficient credentials to see the content of another user, a warning message is displayed (Fig. 5.5 (d)). A more detailed description of our demo consisting of the description of the user credentials, their XML encoding, and an example sequence of messages being exchanged in the system can be found at our web site: *http://dies.cs.utwente.nl/~czenkom/tulip/doc*.

We are proud to say that we implemented our TuLiP system in relatively short time. Even though the time needed to implement the LIAR algorithm is hard to measure precisely (as we were constantly improving the algorithm and fixing the bugs), the basic version of the LIAR algorithm was implemented in approximately 2 weeks. For the rest of the system we needed approximately 6 weeks (here we would like to emphasise that the author of this thesis was the only programming force). Table 5.1 lists different system components, the languages used, and how many lines of code (including comments) contains the corresponding implementation.

5.5.1 The Threat Model

In a real world setting we have neither error-free designs (because they tend to be too complex) nor perfectly secure systems (because that would be too costly). Standard TuLiP is not an exception here, and in this section we look closer at the situations when something is most likely to go wrong.

Standard TuLiP is a distributed system (Fig. 5.1). At any given time there will be many instances of Standard TuLiP components operating independently from each other and there is no centralised authority with a complete overview over what happens in the system. For instance, in a system with millions of users millions of the credential servers would be active

Component	Language	Lines of Code
Front-End	Action Script (Flash), HTML, CSS	562
uiserver	Python	1215
Query Server	Python	1307
LIAR	Prolog (YAP)	737
Credential Requester	Python	1117
Credential Server	Python	1192
Mode Register	HTTP, PHP, CSS, MySQL	3132
Mode Checker	Python	666
Other utilities	Python	285
	Total:	10213

Table 5.1: Components and the corresponding implementation language and programming effort in lines of source code

at any given moment with hundreds of thousands of LIAR engines being engaged in evaluating as many queries. In such a highly distributed system one should consider many possible threats at different levels of abstraction: from the low level problems like reliability of the network connection or stability of each particular system component, to higher level security threats resulting from the malicious activity of an adversary. Although by choosing an appropriate security technology many of the threats can be avoided (at least to some extent), one still should consider situations in which some system components are controlled by a malicious user. A malicious user - an adversary - may attack any part of the system: starting from the network connection and ending with more sophisticated techniques that would result in the attacker having full control over the user's account.

A complete analysis of all possible threats in Standard TuLiP is outside the scope of this thesis. In our opinion such an analysis should be carried out for a production code and would be rather superficial for the proof of concept implementation that we provide. In particular, as mentioned in Sect. 5.3 we assume that an appropriate implementation level mechanism can be used to guarantee the confidentiality, integrity, and non-repudiation in the system. In this section therefore we focus on high level security threats so that we can demonstrate the intrinsic security properties of Standard TuLiP in face of the malicious activity of an adversary regardless of the underlying technology used.

We present an analysis for the two most prominent usage scenarios: issuing a credential, and using the system to evaluate an access control query. This distinction is helpful because different system components play a different role in each of this two scenarios. For instance, the mode register is normally not required during the evaluation of a query, but may be needed if the process of issuing a credential was disturbed. On the other hand, the LIAR engine is crucial during the evaluation of a query, but it is not used during the process of issuing a credential. In our analysis we refer to the high level architecture of Standard TuLiP as given in Fig. 5.1.

Issuing a credential In issuing a credential the following components are used (Fig. 5.1): the User Client, the Mode Register, and one or more Credential Servers. The LIAR engine is not used when issuing a credential. We say that a credential is properly issued

if (1) all credential atoms occurring in the credential are assigned the right modes and (2) the credential is deposited in the right credential server. At a higher level we identify the following two threats:

1. the Mode Register provides the wrong modes for the credential atoms in the credential,
2. the credential is uploaded to the wrong Credential Server.

Abstracting away from the low level implementation problems (like network failures), the first threat above can result from two different situations: (a) the User Client application may be compromised and (b) the Mode Register becomes compromised and returns the wrong modes for the credential atoms. The situation in which the User Client application is compromised (a) is less interesting here, as it implies that the adversary probably already has access to the user's machine and therefore the user's TuLiP account. In this case the adversary can masquerade as the user, and Tulip cannot prevent any malicious actions of the attacker. Acquiring a compromised version of the User Client application by the user can be avoided by performing an integrity check (see for example *GnuPG* [51]) on the application binaries. An attack on the Mode Register (b) is more subtle and therefore more interesting. Recall that the User Client application receives from the Mode Register the modes of the credential atoms for each newly issued credential. Because in Standard TuLiP the modes assigned to the credential atoms in a credential determine the depositary of this credential, by returning invalid modes from a compromised mode register, the adversary influences the storage location for newly issued credentials. The following example demonstrates what may happen when the adversary can manipulate the modes.

Example 5.3 Assume that *eStore* wants to issue a credential allowing all the students from each accredited university to receive a discount. Then let us assume that the valid (i.e. non-compromised) mode values for the credential atoms *discount*, *accredited*, and *student* are: $mode(discount) = ii = (In, In)$, $mode(accredited) = io = (In, Out)$, and $mode(student) = oi = (Out, In)$. In this configuration the *discount* credential would be stored by the *eStore* (because *eStore* is the issuer), the *accredited* credentials would be stored by the accreditation board (*accBoard* in our case), and each student would store her *student* credential. If the Mode Register is compromised, the adversary may freely change the modes of the credential atoms. Assume that the modes returned by the compromised mode register are as follows: $mode(discount) = ii = (In, In)$, $mode(accredited) = oi = (Out, In)$, and $mode(student) = oi = (Out, In)$. What is different in this configuration is that now each *accredited* credential is expected to be stored by the accredited university. Summarising, the credential issued and stored by the *eStore* has the following form:

$$C_1 : \textit{discount\_ii}(\textit{eStore-pub-key}, X) \longleftarrow \textit{student\_oi}(Y, X), \textit{accredited\_oi}(\textit{accBoard-pub-key}, Y).$$

This credential is traceable w.r.t. the mode values returned by the compromised mode register, but *is not* traceable w.r.t. the actual mode values stored in the original mode register (i.e. $mode(discount) = ii$, $mode(accredited) = io$, and $mode(student) = oi$). Now assume that *alice*, who is a student of the University of Twente (*ut*) issues query:

$$Q : \longleftarrow \textit{discount\_ii}(\textit{eStore-pub-key}, \textit{alice-pub-key}).$$

The University of Twente is an accredited university. Because in the proper mode assignment, $\textit{mode}(\textit{accredited}) = \textit{io} = (In, Out)$, the accreditation board (*accBoard*) stores the credential:

$$C_2 : \textit{accredited\_io}(\textit{accBoard-pub-key}, \textit{ut-pub-key}).$$

Now, when query Q is received by the LIAR engine, LIAR will fetch credential C_1 (which is not traceable w.r.t. to the original mode assignment) and LIAR will search for the *accredited* credential at the credential server corresponding to the University of Twente rather than at the server of the accreditation board *accBoard*. This is because the mode of the *accredited* credential atom has been changed from (In, Out) to (Out, In). Beacuse *ut* does not store credential C_2 (it is stored by *accBoard*), LIAR will return negative answer to the query. It means that *alice* will not receive the discount she is eligible for.

Example 5.3 shows that by manipulating the modes in a compromised mode register the adversary can disable some services available to the user. This can be regarded as a subtle version of the *deny-of-service* attack. An additional observation from Example 5.3 is that the compromised mode register can influence only the newly issued credentials. In Example 5.3, if the right version of credential C_1 was already stored at the credential server of *eStore* then the answer to query Q would be positive and *alice* would receive the discount as expected.

In Example 5.3, a user is not given authorisation she has rights to. A much worse situation would be if a user who is not authorised to perform an action on a resource actually receives the authorisation. For this to happen it is not sufficient to break the mode register. Even when the Mode Register is broken, this has only influence on the newly issued credentials. It means that, in order to gain unauthorised access to a resource or a service, one has to be able not only to have control over where the credential get stored and have access to the right credential server, but also have to be able to issue new credentials on behalf of another user. This can happen if the public identifier of a user becomes compromised, which means that the adversary has full control over the user's account.

An interesting situation occurs when the mode register is not compromised (i.e. the mode register returns valid mode values), but the adversary has full control over the user's account.

Example 5.4 Assume the following state:

eStore:

(1) $\textit{discount\_ii}(\textit{eStore}, X) \longleftarrow \textit{accredited\_io}(\textit{accBoard}, Y), \textit{student\_ii}(Y, X).$

accBoard:

(2) $\textit{accredited\_io}(\textit{accBoard}, \textit{ut}).$

ut:

(3) $\textit{student\_ii}(\textit{ut}, \textit{alice}).$

In this scenario each credential is stored by the issuer. As a consequence, even when having full control over the account of another user, the adversary cannot receive authorisation which was not already given to the user. In our example, if credential (3) does not exist, the adversary will not have discount at the *eStore* despite having full control over *alice*'s account.

Example 5.4 shows, that by careful selection of the modes the bad consequences of an attack can be limited.

Evaluating a query When evaluating a query, the use of the Mode Register is not required. Unless other system components are also compromised, the system remains consistent as long as no new credentials are added to the system even if the Mode Register is broken. Still, the use of the mode register during the query evaluation can be considered in order to dynamically validate the modes occurring in a credential.

Assuming that the mode register is not being used during the query evaluation, the adversary may target the following system components: the User Client, the LIAR engine, and the Credential Servers.

If the adversary manages to make the user work with a compromised User Client application, the adversary can quickly gain full access to the user's machine and therefore take full control over the user's TuLiP account. As before, this attack can be avoided by performing an integrity check on the on the application binaries and by making sure that the user's computer is free of malicious code.

The LIAR engine and user's credential server might be run on the user's machine or can be considered part of the underlying infrastructure. In the first case, the integrity of the LIAR engine (resp. a credential server) must be protected in the same way as the integrity of the User Client application. A compromised LIAR engine (resp. credential server) leads to the same problems as a compromised User Client application: a compromised user public identifier and the full control of the user's account by the adversary.

In the case when the LIAR engine and the credential servers are part of the underlying infrastructure, the integrity of the system directly depends on the security of the underlying infrastructure. In Section 5.3.3 we show how Distributed Hash Tables (DHT)[85] can be used in Standard TuLiP to provide the mapping between a user's public identifier and the network address of the credential server corresponding to this user. In this approach the credential server is part of the infrastructure, and the user does not have to run an instance of a credential server on her machine. The same approach can be used for the deployment of a LIAR engine. In this approach, besides running an instance of a credential server, each DHT node also runs an instance of the LIAR engine. In this case the LIAR engine used to evaluate a query may be determined by the identifier of the query issuer or the issuer of the first credential atom in the query (other approaches are also possible). As a consequence, the user does not know which LIAR engine is used to evaluate the query. This makes access to the LIAR engine more difficult also to the adversary.

5.6 Related Work

We present the general related work on trust management in Chapter 2 and the related work specific to Core TuLiP in Chapter 4. In this section we emphasise the practical and implementation related aspects of the most important trust management systems.

The idea of binding a public key to the action this public key is trusted to perform is already present in the pioneering work on Decentralised Trust Management by Blaze, Feigenbaum and Lacy [27]. Here we follow this approach by using public keys as the user identifier.

Similar approach can also be found in the SDSI/SPKI system [36]. As already mentioned in Chapter 2, the credential discovery is not covered by these works.

In RT [66, 67] the problem of common vocabulary is handled by the means of the XML namespaces. The authors propose the so called *Application Domain Specification Document* (ADSD) to hold the role names and the related information like storage type or a natural-language description of these role names. In ADSD each role name contains a vocabulary id, which can be the URI of the ADSD, and the id of the role name in this ADSD. ADSDs allow issuers to agree on the common vocabulary and storage types. Because in RT each issuer can have its own ADSD, this approach seems to be more "distributed" than ours. However, in Standard TuLiP we could allow for multiple mode registers as well, and then, for example, include the URI of the mode register defining the given role name in the *uri* attribute of the *rolename* element. We believe, however, that such a decentralisation makes system hard to manage and error-prone as it is hard to guarantee that all the systems storing ADSDs will be online when one needs them. In our approach the mode register is used when writing credentials and is normally not needed during query evaluation.

Trust-$\mathcal{X}$, introduced by Bertino et al. [22], uses the $\mathcal{X}$-TNL trust negotiation language for expressing credentials and disclosure policies. Trust-$\mathcal{X}$ certificates are either credentials or declarations. Credentials state personal characteristics of the owner and are certified by a Credential Authority (CA). Declarations also carry personal information about its owner but are not certified. Trust-$\mathcal{X}$ is then closer to the traditional authorisation mechanisms based on identity-based public-key systems like X.509.

The *eXtensible Access Control Markup Language* (XACML) [77] supports distributed policies and also provides a profile for the role based access control (RBAC). However, in XACML, it is the responsibility of the *Policy Decision Point* (PDP) – an entity handling access requests – to know where to look for the missing attribute values in the request. The way missing information is retrieved is application dependent and is not directly visible in the supporting language.

5.7 Conclusions

In this chapter we present Standard TuLiP - a logic based trust management system. Standard TuLiP can be seen as a practical realisation of Core TuLiP which we present in Chapter 4 and which is a full-fledged trust management system. The basic constituents of Standard TuLiP are the Standard TuLiP trust management language, the mode register for managing role names and the associated modes, a set of credential servers, where users store their credentials, and a terminating sound and complete Lookup and Inference AlgoRithm (LIAR) which guarantees that all required credentials can be found when needed.

Standard TuLiP is decentralised. Every user can formulate his/her own security policy and store credentials in the most convenient and efficient way for himself. Standard TuLiP does not require a centralised repository for credential storage, nor does it rely on any external PKI infrastructure. Standard TuLiP credentials are signed directly by their issuers so that no preselected Certification Authority (CA) is needed.

With this we show that it is possible to design and implement a trust management system that is theoretically sound yet possible to deploy in practice.

CHAPTER 6

TuLiP
Logic Programming for
Trust Management

In Chapter 4 we present Core TuLiP - the theoretical foundations for the TuLiP trust management system. There we introduce the basic syntax of the language, show the declarative semantics, and show the use of modes in planning the credential distribution and in the credential discovery. Finally, we define the Lookup and Inference AlgoRithm (LIAR) and prove that LIAR is sound and complete wrt the declarative semantics. Then, in Chapter 5 we investigate practical aspects of deploying Core TuLiP. We look at the credential encoding, multiple storage options, and we define the necessary system components. Finally, we discuss the system from the perspective of a user: how to create a credential, how to issue a query, what problems can be encountered when the user identifier becomes compromised, and how to protect the system against such problems. In this chapter, we combine our experience and knowledge from the previous chapters and we finally present TuLiP - a trust management system based on logic programming. Here we formalise the notions informally introduced in the previous chapters like redundant storage or (user-defined) constraints. Finally, we also improve the declarative reading of the language by separating the meaning of a credential from the way a credential should be deployed.

6.1 Introduction

In this chapter we present TuLiP - a Trust management system based on Logic Programming. TuLiP is built on the theory we present in Chapter 4 - Core TuLiP. The syntax of Core TuLiP is restricted: a credential atom can have only two arguments - namely the issuer and the subject. A credential in Core TuLiP is assumed to be stored by only one entity, which is determined by the mode assigned to a given credential atom. Moreover, Core TuLiP supports only built-in constraints. To some extent, we lift these restrictions in Standard TuLiP (Chapter 5), where we add an additional XML argument to a credential atom and we allow a credential to have more than just one depositary. The extensions presented in Chapter 5 have a practical dimension and so they are neither defined formally nor fully discussed. For instance, the mapping between the XML content contained in the XML element *properties* of the Standard TuLiP credential and the corresponding logic programming term is not formally defined. Standard TuLiP also supports only built-in constraints.

In this chapter we address the above mentioned issues. First, we extend the syntax of the language: we allow for an unbounded number of arguments, each of which can be either a standard Prolog term or a so called Prolog Markup Language (PML) term representing actual XML content. Next, we deal with the constraints in TuLiP. We define the built-in and the user-defined constraint, we set out the requirements for the constraints evaluation algorithm, and we introduce the notion of a TuLiP package.

In order to handle redundant storage in a formal way we introduce the notion of a *bound* credential. Bound credentials allow us to separate the intended meaning of a credential from where the credential is stored. This is the solution to the problem of the changing order of the body atoms depending on the mode assignment (see Chapter 4, Section 4.3, Example 4.3).

To accommodate these extensions, we refine the notion of the depositary and the traceable credential. We also extend our LIAR algorithm appropriately and we show the updated proofs of the soundness and completeness of the extended algorithm.

In this chapter, we also show how to map an XML content to a Prolog term and vice versa.

The chapter is structured as follows. In Section 6.2 we present the extended syntax of the language and we update the definition of the depositary and the traceable credential. In Section 6.3 we formally define built-in and user-defined constraints. Section 6.4 deals with the redundant storage. Here we introduce bound atoms and credentials. In Section 6.5 we present the extended LIAR algorithm, we show an example, and then we discuss the declarative semantics and soundness and completeness results. Finally, in Section 6.6, we introduce *Prolog Markup Language* (PML). We finish the chapter with the Related Work (Section 6.7) and the Conclusions (Section 6.8).

6.2 TuLiP

We start the description of the TuLiP trust management system by presenting its language. The language of TuLiP is an extension of Core TuLiP (Chapter 4) and Standard TuLiP (Chapter 5) in that a credential is definite clause $H :- B_1, \ldots, B_n$ where H is a *credential atom* and each B_i can be either a credential atom or a constraint. The difference between TuLiP and Core TuLiP (and Standard TuLiP) is in the definition of a credential atom and a con-

straint (we discuss the constraints in next section). In Core TuLiP, a credential atom has two arguments: the *issuer* and the *subject*. A Standard TuLiP credential has three arguments: the *issuer*, the *subject* and the *properties* where the *properties* argument can hold additional attributes of the issuer and/or the subject and is a Prolog Term representing XML content (we present how XML content maps to a Prolog term in Section 6.6). The mode of the *properties* argument is fixed to be Out. In TuLiP we relax this limitation by allowing a credential atom to have any number of arguments and more flexible mode assignments. Summarising, in TuLiP, a *credential atom* has the following form:

$$\mathit{rolename}(\mathit{issuer}, \mathit{subject}, \mathit{arg}_1, \ldots, \mathit{arg}_n).$$

where $n \geq 0$. Both the *issuer* and the *subject* must be either a ground term or a logical variable. Each credential argument arg_i may be an arbitrary Prolog term.

Adding additional arguments to a credential atom might seem to be a straightforward syntactic extension, but this is not so. Adding additional arguments to a credential atom affects the discovery of the *subject-traceable* credentials. The following example demonstrates the problem:

Example 6.1 Take the following set of credentials:

$(1)\ r(a, X, Y) \longleftarrow r_1(b, X, Y).$
$(2)\ r_1(b, X, Y) \longleftarrow r_2(c, X, Y).$
$(3)\ r_2(c, d, X) \longleftarrow X == \mathit{attr}.$

and assume that all predicate symbols have mode (In, In, In). The $==/2$ is a built-in constraint which takes two terms as arguments and succeeds if they are strictly equal. For the given mode assignment, credential (1) is stored by entity a, credential (2) by entity b and credential (3) by entity c. Take the query $Q = r(a, d, \mathit{attr})$. Applying the LIAR algorithm we use in Standard TuLiP to this query would result (after a few iterations, working top-down) in the following clauses being added to CLSTACK:

$r(a, d, \mathit{attr}) \longleftarrow r_1(b, d, \mathit{attr}).$
$r_1(b, d, \mathit{attr}) \longleftarrow r_2(c, d, \mathit{attr}).$
$r_2(c, d, \mathit{attr}) \longleftarrow \mathit{attr} == \mathit{attr}.$

When $==(\mathit{attr}, \mathit{attr})$ is selected as a new goal, because $==/2$ is a built-in constraint, $==(\mathit{attr}, \mathit{attr})$ can be immediately evaluated and added to FACTSTACK. Then, in Phase 2 of the algorithm, atoms $r_2(c, d, \mathit{attr})$, $r_1(b, d, \mathit{attr})$, and $r(a, d, \mathit{attr})$ are added to FACTSTACK. The answer to the query is - as expected - positive.

Now, if we change the modes of all predicate symbols above to (Out, In, In), credential (1) is stored by entity b, credential (2) by entity c, and credential (3) by entity d. Now the result of the evaluation of query $Q = r(a, d, \mathit{attr})$ is different. LIAR first fetches from entity d all credentials for which the mode of the first (*issuer*) argument is Out and the mode of the second (*subject*) argument is In. In our example, LIAR fetches credential (3) (in unchanged form) and adds it to CLSTACK. Next, working bottom-up, LIAR fetches credential (2) from c and credential (1) from b. At this point, the contents of CLSTACK is the following:

$r(a, X, Y) \longleftarrow r_1(b, X, Y).$

$r_1(b, X, Y) \longleftarrow r_2(c, X, Y).$
$r_2(c, d, X) \longleftarrow X == attr.$

In this case, however, no well-moded goal can be selected from any clause in the CLSTACK. This is because the unification $X \rightarrow attr$ cannot be performed when discovering credentials bottom-up (see Chapter 4) and the variable X in credential (3) will never be unified with *attr* from the query. As a consequence, the evaluation of the query Q returns negative answer.

To cope with this problem we need to modify the requirements for the traceable credential and update Definition 4.3.2 from Chapter 4 (here, the mode of an atom A is denoted by m_A).

Definition 6.2.1 (Traceable, Depositary) *Let $cl = H :- B_1, \ldots, B_n$, $n \geq 0$, be a well-formed credential. We say that cl is* traceable *if the following conditions hold:*

- $\forall i \in [1, n]$, *if* $m_{B_i}(\text{issuer}) = Out$ *and* $m_{B_I}(\text{subject}) = In$ *then for each argument* arg_j *of* B_i, $arg_j \in Out(B_i)$,
- *if* $m_H(\text{issuer}) = In$, *then* $\text{depositary}(cl) = \text{issuer}(H)$,
- *if* $m_H(\text{issuer}) = Out$ *and* $m_H(\text{subject}) = In$ *then:*
  - *for each argument* arg_i *of* H, $arg_i \in Out(H)$,
  - *if* $\text{subject}(H)$ *contains a ground term then* $\text{depositary}(cl) = \text{subject}(H)$,
  - *if* $\text{subject}(H)$ *contains a variable, then there exists a prefix* $B_1, \ldots, B_k$ *of the body such that:*
    - $\forall i \in [1, k]$, $m_{B_i}(\text{issuer}) = Out$ *and* $m_{B_i}(\text{subject}) = In$,
    - $\text{subject}(H) = \text{subject}(B_1)$,
    - $\text{subject}(B_{i+1}) = \text{issuer}(B_i)$, *and is a variable, for* $i \in [1, k-1]$,
    - $\text{issuer}(B_k)$ *contains a ground term,*

    and $\text{depositary}(cl) = \text{issuer}(B_k)$.

Definition 6.2.1 says that if there is a credential atom in a credential such that the mode of the issuer is Out and the mode of the subject is In then each argument of this credential atom must have mode Out.

6.3 Constraints

A TuLiP credential can have constraints in the body. Informally, a constraint in TuLiP is any atom which is not a credential atom. A constraint can be either a built-in constraint, or a user-defined constraint. A built-in constraint is available in any TuLiP system and is embedded directly into the decision algorithm LIAR. A user-defined constraint, on the other hand, can be defined by the user. In this section we address the following issues: (1) what is in the definition of a user-defined constraint, (2) what is the mode associated with a user-defined constraint, (3) who is the depositary of a user-defined constraint, and (4) how a user-defined constraint is evaluated.

We start with an example showing user-defined constraints.

Example 6.2 A company has a private database storing the information about each employee. Some data in the employee database is confidential and should not be visible to a company partner or some other external entity. For this reason, before showing the data of an employee, the company filters the data to check which data can and which cannot be revealed (in other words the company checks whether the request comply with the company privacy policy).

(1) *employee*$(comp, X, Y) \longleftarrow$ *employee_int*$(comp, X, Y2)$, *filter*$(Y2, Y)$.
(2) *filter*$([\,], [\,])$.
(3) *filter*$([AttName : AttValue|Atts], Atts2) \longleftarrow$ *confidential*$(AttName)$, *filter*$(Atts, Atts2)$.
(4) *filter*$([AttName : AttValue|Atts], [AttName : AttValue|Atts2]) \longleftarrow$
non-confidential$(AttName)$, *filter*$(Atts, Atts2)$.
(5) *confidential*$(salary)$.
(6) *confidential*$(bankaccount)$.
(7) *non-confidential*$(position)$.
(8) *employee_int*$(comp, marcin, [salary:2500,position:phd,bankaccount:123456])$.

From the clauses above only two clauses are credentials: clause (1) and clause (8). We have then two credential atoms: *employee*$/3$ and *employee_int*$/3$. All the remaining atoms are therefore constraints. We see that each constraint must have its *definition*. For instance the definition of constraint *filter*$/2$ consists of credentials (2), (3), and (4). A constraint clause may contain other (built-in or user-defined) constraints, but it must not contain a credential atom. For instance in constraint clause (3) above, *confidential*$(AttName)$ is another user-defined constraint.

In Example 6.2 in order to keep the policy consistent and complete, for each attribute one needs to remember to define the attribute as either *confidential* or *non-confidential*. If an attribute is neither *confidential* nor *non-confidential* (simply because one forgot to add an appropriate definition) the *filter*$/2$ will fail for each non-empty attribute list making *employee*$/3$ failing as well. If the policy is inconsistent (an attribute is defined to be *confidential* and *non-confidential* at the same time) one will receive two answers for each inconsistent attribute: one including the inconsistent attribute and one not including the inconsistent attribute.

Keeping the policy consistent could be made easier if one may use negation in the policy. In such a case one can simply write:

(3) *filter*$([AttName : AttValue|Atts], Atts2) \longleftarrow not(public(AttName))$, *filter*$(Atts, Atts2)$.

Now it is sufficient to only say that an attribute is *public*, otherwise an attribute will be treated as *confidential*.

How constraints are distinguished from the credential atoms is implementation dependent - for instance, if a constraint is encoded in XML language, then the corresponding XML document may have an XML element or attribute specifying whether the given element is a constraint or a credential atom.

Definition 6.3.1 (Constraints, Constraint Clauses) *A constraint is a predicate symbol having zero or more arguments which is not a credential atom. A* constraint clause *is a definite clause of the form* $C_0 \longleftarrow C_1, \ldots, C_n., n \geq 0$ *in which each* C_i *is a constraint. The set of constraint clauses defining a constraint is called the* definition *of this constraint.*

Because constraints are used in credentials, we require each constraint to be moded, and we require each constraint clause to be well-moded. In contrast to a credential atom, the mode of a constraint does not determine where the defining constraint clauses are stored.

Example 6.3 Returning to Example 6.2 assume the following mode assignment:

$$\begin{aligned} \mathit{mode}(\mathit{employee}) &= (\mathit{In}, \mathit{In}, \mathit{Out}),\\ \mathit{mode}(\mathit{employee\_int}) &= (\mathit{In}, \mathit{In}, \mathit{Out}),\\ \mathit{mode}(\mathit{filter}) &= (\mathit{In}, \mathit{Out}),\\ \mathit{mode}(\mathit{confidential}) &= (\mathit{Out}),\\ \mathit{mode}(\mathit{non\text{-}confidential}) &= (\mathit{Out}). \end{aligned}$$

For this mode assignment, credential (1) and credential (8) are stored by *comp*. But where are the constraint clauses $(2) - (7)$ stored ? If LIAR needs to evaluate credential (1), LIAR needs all the remaining clauses. Clause (8) is fetched by the means of the standard discovery process (described in Chapter 4). For the constraint clauses, the most intuitive approach (and the approach we decided to take) is to fetch all the constraint clauses required to evaluate credential (1) at the same time when credential (1) is being fetched. So, in TuLiP, credential (1) and constraint clauses $(2) - (7)$ are fetched at the same time (see Section 6.3.1 where we introduce the notion of a *package*). We can say therefore, that for each constraint clause $(2) - (7)$, the depositary of this constraint clause is *comp*. We also say that *comp* is the issuer for each of the constraints $(2) - (7)$.

Definition 6.3.2 (Depositary of a Constraint Clause) *Let cl be a credential and let C be a constraint occurring in the body of cl. Let cl_C be a constraint clause defining C. Then:*

- issuer$(cl_C) =$ issuer(cl)*, and*
- depositary$(cl_C) =$ depositary(cl).

The consequence of Definition 6.3.2 is that if the credential containing a constraint C is stored at more than one location, then each constraint clause defining C will be duplicated at each of these locations. In other words, a constraint definition follows the credential containing the constraint.

In Example 6.2, the definition of each constraint is given be a Prolog program. As the following example shows, this is not always the case.

Example 6.4 In a medical corporation *mediCorp*, a physician may read a medical record of a patient only if she is the designated primary care physician of this patient. In TuLiP this can be expressed as follows:

$$\begin{aligned} \mathit{readRecord}(\mathit{mediCorp}, X, P, R) \longleftarrow\ & \mathit{retrieveFromDatabase}(P, R),\\ & \mathit{extractField}(R, \textit{'desig-physician'}, X). \end{aligned}$$

Here both *retrieveFromDatabase*(P, R) and *extractField*$(R,$ *'desig-physician'*$, X)$ are user-defined constraints. The former makes a connection to a company database and retrieves the medical record file R of patient P. The latter takes document R and extracts the value of field *'desig-physician'*. R can be an XML document but can be also any other document in

a proprietary document format used by *mediCorp*. In the case the database does not contain any record for the given patient or the retrieved document does not contain the requested field, the corresponding predicate fails.

In this example the definition of *retrieveFromDatabase*(P, R) and *extractField*$(R,$ *'desig-physician'*$, X)$ is given in some external module. We call such a module an *evaluation algorithm*. This algorithm must be known to LIAR in order to evaluate *retrieveFromDatabase*(P, R) or *extractField*$(R,$ *'desig-physician'*$, X)$. In general, LIAR needs to be provided with an appropriate evaluation algorithm for every user-defined constraint.

The entity which defines a constraint must provide an evaluation algorithm for this constraint. There is a lot of freedom in designing the evaluation algorithm for a constraint. However, some restrictions apply:

Definition 6.3.3 (Requirements for the Evaluation Algorithm) *Given a constraint C, the evaluation algorithm for C must satisfy the following two conditions:*

(1) the algorithm terminates,

(2) the algorithm returns either FAIL *or a non-empty finite set $\alpha_1, \ldots, \alpha_n$ of ground computed answer substitutions such that for each i:*

(2a) $Dom(\alpha_i) = VarOut(C)$,

(2b) the algorithm respects the modes: if $VarIn(C) = \emptyset$ then $VarOut(C\alpha_i) = \emptyset$.

A constraint evaluation algorithm can be given in terms of a well-moded Prolog program which is then executed by means of the LD-resolution (SLD resolution combined with the leftmost selection rule) in which case we additionally assume that the program terminates for each well-moded query.

6.3.1 Packages

In order to evaluate a constraint, the issuer of this constrain must provide all the related constraint clauses (if written as logic programming rules) or the evaluation algorithm which can be used to evaluate this constraint. The constraint clauses (resp. the evaluation algorithm) is fetched together with the credential that contains the constraint. This brings us to the notion of a TuLiP *package*.

Definition 6.3.4 (Package) *Given a credential cl, a* package *corresponding to this credential consists of (1) credential cl and (2) a set of constraint clauses defining each constraint appearing in credential cl* or *a constraint evaluation algorithm capable of evaluating each of the constraints appearing in credential cl.*

We use $cl : \zeta$ to denote a package for credential cl where ζ represents a set containing the constraint clauses *or* a constraint evaluation algorithm. If C is a constraint occurring in cl then the process of evaluating constraint C is denoted by $\zeta(C)$. The issuer of package $cl : \zeta$ is the issuer of credential cl. Similarly, the depositary of package $cl : \zeta$ is the depositary of credential cl.
In other words, a package contains a credential and everything else which is necessary to evaluate all the constraints occurring in this credential.

We conclude this section with an example.

Example 6.5 *For the Example 6.2 we have the following package* $cl : \zeta$*:*

- cl*:*

  (1) *employee*$(comp, X, Y) \longleftarrow$ *employee_int*$(comp, X, Y2)$, *filter*$(Y2, Y)$.

- ζ*: the set of constraint clauses:*

  (2) *filter*$([\,], [\,])$.
  (3) *filter*$([AttName : AttValue|Atts], Atts2) \longleftarrow$
  confidential$(AttName)$, *filter*$(Atts, Atts2)$.
  (4) *filter*$([AttName : AttValue|Atts], [AttName : AttValue|Atts2]) \longleftarrow$
  non-confidential$(AttName)$, *filter*$(Atts, Atts2)$.
  (5) *confidential*$(salary)$.
  (6) *confidential*$(bankaccount)$.
  (7) *non-confidential*$(position)$.

6.4 Multiple Modes and Redundant Storage

Sometimes we want to store credentials at more than just one location. For instance, one may need to protect the system from node failures, or overcome the problem of authorities going off line or being unreachable. Additionally, having more than just one storage configuration can allow the system deployer to better balance the network load.

For these reasons, in Standard TuLiP (Chapter 5) we allow a credential to have more than one depositary. We do so by assigning multiple modes to a credential atom, which allows us to store a credential at the issuer and, at the same time, at the subject or some other entity. For instance, if *student*$(ut, alice)$ is a credential, and *student*$/2$ has modes (In, In) and (Out, In) then *student*$(ut, alice)$ will be stored at both *ut* and *alice*.

Recall from Chapter 4 that the mode of a credential atom may influence the order of the atoms in the body of a credential. This is the consequence of requiring that each credential must be well-moded. By allowing multiple modes to be assigned to one credential atom, the order of the atoms in the body of a credential may be different for each selected combination of modes. The following example illustrates this:

Example 6.6 Consider the following credential:

$$(1)\textit{discount}(\textit{eStore}, X) \longleftarrow \textit{accredited}(\textit{accBoard}, Y), \textit{student}(Y, X).$$

and assume that the mode register has the following mode assignments:

$$\begin{aligned} \textit{mode}(\textit{discount}) &= \{(In, In), (Out, In)\} \\ \textit{mode}(\textit{accredited}) &= \{(In, Out), (Out, In)\} \\ \textit{mode}(\textit{student}) &= \{(In, In), (Out, In)\} \end{aligned}$$

Recall that for the credential to be traceable, for every mode value of the head, there must exist a permutation of the credential atoms occurring in the body and the corresponding mode

values such that the credential is well-moded. For the credential above one can choose the following two *versions* of the credential above:

$$(2)\ \ \mathit{discount}(\mathit{eStore}, X)^{(In,In)} \longleftarrow \mathit{accredited}(\mathit{accBoard}, Y)^{(In,Out)}, \mathit{student}(Y, X)^{(In,In)}.$$

$$(3)\ \ \mathit{discount}(\mathit{eStore}, X)^{(Out,In)} \longleftarrow \mathit{student}(Y, X)^{(Out,In)}, \mathit{accredited}(\mathit{accBoard}, Y)^{(Out,In)}.$$

Here, the superscript denotes the mode associated with the given credential atom. For credential (3) above, we had to change the order of the atoms in the body in order to keep the credential well-moded. We call credentials (2) and (3) *bound credentials* and the original credential without fixed modes (1) an *unbound credential* or simply a *credential* if clear from the context.

Example 6.6 shows that when multiple modes are allowed one needs to choose not only the storage (by selecting appropriate modes) but one may also need to change the order of the atoms in the body in order to keep the credential well-moded. It would be much easier to select the desired storage options and let the system produce the required corresponding bound version of a credential automatically. For this reason we introduce the notion of a bound credential.

In the remaining part of this section we formalise the concept of bound credentials. We begin with the definition of a *mode set*. Recall, that in Standard TuLiP, the modes are stored in a *mode register*. A *mode set* is its theoretical equivalent.

Definition 6.4.1 (ModeSet) *Let Pred be a set of predicate symbols. A* mode set $\mathcal{M}$ *for Pred is a mapping that maps every predicate symbol $p/n \in$ Pred to a set of modes. Given a predicate symbol p/n, we denote the set of modes assigned to p/n by $\mathcal{M}(p/n)$.*

Now we can introduce the notion of *bound* atoms and credentials.

Definition 6.4.2 (Bound Atoms, Bound Credentials) *Let A be an atom (a credential atom or a constraint). We call A^{m_A} a* bound atom *if $m_A \in \mathcal{M}(A)$. A credential $H^{m_H} \longleftarrow A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$, in which H^{m_H} and each $A_i^{m_{A_i}}$ are bound atoms, is called a* bound credential. *A* bound query *is a sequence of bound atoms.*

Definition 4.2.2 of well-moded clauses from Chapter 6 applies directly to bound credentials:

Definition 6.4.3 (Well-Moded Bound Credentials and Queries)
Let $cl = H^{m_H} \longleftarrow A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$ be a bound credential. We say that cl is well-moded *if $\forall\, i \in [1, n]$ the following holds:*

$$\mathit{VarIn}(A_i^{m_{A_i}}) \subseteq \bigcup_{j=1}^{i-1} \mathit{VarOut}(A_j^{m_{A_j}}) \cup \mathit{VarIn}(H^{m_H}), \text{ and}$$

$$\mathit{VarOut}(H^{m_H}) \subseteq \bigcup_{j=1}^{n} \mathit{VarOut}(A_j^{m_{A_j}}) \cup \mathit{VarIn}(H^{m_H}).$$

A bound query $A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$ is well-moded iff $H^{m_H} :- A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$, where H^{m_H} is a (dummy) bound atom with zero arity, is a well-moded bound credential.

Similarly, we provide "bound" versions of Definition 4.3.1 (Well-Formed), and Definition 6.2.1 (Traceable, Depositary).

Definition 6.4.4 (Well-Formed Bound Credentials) *Let* $cl = H^{m_H} \longleftarrow A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$ *be a bound credential. We say that* cl *is* well-formed *if it is well-moded and* issuer(H) *is a ground term.*

Definition 6.4.5 (Bound Traceable, Depositary) *Let* $cl = H^{m_H} \longleftarrow A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$, $n \geq 0$*, be a well-formed credential. We say that* cl *is* traceable *if the following conditions hold:*

- $\forall i \in [1, n]$*, if* $m_{A_i}(\text{issuer}) = Out$ *and* $m_{A_i}(\text{subject}) = In$ *then for each argument* arg_j *of* A_i*,* $arg_j \in Out(A_i)$*,*
- *if* $m_H(\text{issuer}) = In$*, then* depositary$(cl) =$ issuer(H)*,*
- *if* $m_H(\text{issuer}) = Out$ *and* $m_H(\text{subject}) = In$ *then:*
  - *for each argument* arg_i *of* H*,* $arg_i \in Out(H)$*,*
  - *if* subject(H) *contains a ground term then* depositary$(cl) =$ subject(H)*,*
  - *if* subject(H) *contains a variable, then there exists a prefix* $B_1, \ldots, B_k$ *of the body such that:*
    - $\forall i \in [1, k]$*,* $m_{B_i}(\text{issuer}) = Out$ *and* $m_{B_i}(\text{subject}) = In$*,*
    - subject$(H) =$ subject(B_1)*,*
    - subject$(B_{i+1}) =$ issuer(B_i)*, and is a variable, for* $i \in [1, k-1]$*,*
    - issuer(B_k) *contains a ground term,*

    and depositary$(cl) =$ issuer(B_k)*.*

Having defined bound (credential and constraint) atoms and bound credentials, and knowing when a bound credential is traceable, we are now interested in the relationship between a non-bound credential and a bound credential. The concept of a *binding* defines this relationship.

Definition 6.4.6 (Binding) *A binding* B *for a credential* $cl = H \longleftarrow B_1, \ldots, B_n$ *is a tuple* $(\pi, m_H, m_{B_1}, \ldots, m_{B_n})$*, where* π *is a permutation* $\{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ *and* $m_H \in \mathcal{M}(H)$*,* $m_{B_1} \in \mathcal{M}(B_1), \ldots, m_{B_n} \in \mathcal{M}(B_n)$*. The result of applying binding* B *to credential* cl *is the bound credential* $cl^B = H^{m_H} \longleftarrow B_{\pi(1)}^{m_{B_{\pi(1)}}}, \ldots, B_{\pi(n)}^{m_{B_{\pi(n)}}}$*.*

The result of applying a binding to a non-bound credential is precisely one bound credential. For this bound credential, we do not know yet if it is traceable. From all bindings available for a credential, only a small fraction of them will "produce" traceable bound credentials. We call this set of bound traceable credentials a *bound set*:

Definition 6.4.7 (Bound Set) *Let* cl *be a non-bound credential. The* bound set *corresponding to* cl*, denoted* BoundSet(cl)*, is defined as follows:*

$$\text{BoundSet}(cl) = \{cl^B \mid B \text{ is a binding for } cl \text{ and } cl^B \text{ is traceable.}\}$$

Given a non-bound credential cl, only the bound traceable credentials from *BoundSet*(cl) are potential candidates for the actual deployment. Here we need to know what is the minimal set of bound traceable credentials (a minimal bound set) which guarantees proper credential discovery. For instance, if a credential atom in the head of a non-bound credential is assigned three different mode values, one should guarantee that a query containing this atom can be answered for each mode value available for this atom. So, if $p/2$ is a credential atom and $\mathcal{M}(p/2) = \{(In, Out), (In, In), (Out, In)\}$ then the bound queries $p/2^{(In,Out)}$, $p/2^{(In,In)}$, and $p/2^{(Out,In)}$ all should yield the correct answer (either negative or positive). Therefore, what we need is a non-bound version of the definition of well-modedness and traceability. Here, we cannot simply reuse the "bound" version of these definitions as they assume one mode value per predicate symbol, whereas in case of non-bound credentials one predicate symbol can be assigned multiple mode values.

Definition 6.4.8 (Well-Moded and Traceable Non-Bound Credentials) *Let $\mathcal{M}$ be a mode-set. We say that non-bound credential $cl = H \longleftarrow B_1, \ldots, B_n$ is* well-moded *(resp.* traceable*) w.r.t. $\mathcal{M}$ if* for each $m_H \in \mathcal{M}(H)$ *there exists a binding B_{m_H} such that $c^{B_{m_H}}$ is a bound credential which has H^{m_H} in the head and is well-moded (resp.* traceable*).*

From Definition 6.4.8 we see that a non-bound credential is well-moded, if for each mode value of the head there exists at least one binding resulting in a well-moded bound credential having the same mode in the head. For a non-bound credential cl to be traceable, Definition 6.4.8 requires that there exists at least one bound traceable credential for each $m_{head(cl)} \in \mathcal{M}(head(cl))$ having the same mode in the head. What happens if the bound set for a non-bound credential is not minimal, i.e. there are more than one bound traceable credential for each mode value of the head ? In such a case, we require that *at least* one of them is chosen and deployed. One can also decide to deploy more than just one of them, which can improve the overall reliability of the system, as "the same" credential is discoverable at more than one node.

Example 6.7 Consider a credential $cl = p(a, X) \longleftarrow q(b, Y), r(Y, X)$. and the following mode sets: $\mathcal{M}(p/2) = \mathcal{M}(q/2) = \mathcal{M}(r/2) = \{(In, In), (In, Out), (Out, In)\}$. This credential is traceable. In fact for each mode of the head there exists at least one traceable binding of the credential having the same mode in the head. For the mode of the head $m_{p/2} = (In, In)$, we have six traceable bindings resulting in the following set of bound traceable credentials:

$$
\begin{array}{ll}
(1) & p(a,X)^{(In,In)} \longleftarrow q(b,Y)^{(In,Out)}, r(Y,X)^{(In,In)}. \\
(2) & p(a,X)^{(In,In)} \longleftarrow q(b,Y)^{(In,Out)}, r(Y,X)^{(In,Out)}. \\
(3) & p(a,X)^{(In,In)} \longleftarrow q(b,Y)^{(In,Out)}, r(Y,X)^{(Out,In)}. \\
(4) & p(a,X)^{(In,In)} \longleftarrow r(Y,X)^{(Out,In)}, q(b,Y)^{(In,In)}. \\
(5) & p(a,X)^{(In,In)} \longleftarrow r(Y,X)^{(Out,In)}, q(b,Y)^{(In,Out)}. \\
(6) & p(a,X)^{(In,In)} \longleftarrow r(Y,X)^{(Out,In)}, q(b,Y)^{(Out,In)}.
\end{array}
$$

From this set of bound credentials one has to choose at least one of them and deploy it according to Definition 6.4.5. For $m_{p/2} = (In, Out)$ and the permutation of the atoms in

the body $q(b, Y), r(Y, X)$, there exists only one traceable binding producing the following bound traceable credential:

$$(7)\ \ p(a, X)^{(In,Out)} \longleftarrow q(b, Y)^{(In,Out)}, r(Y, X)^{(In,Out)}.$$

Similarly, for the mode of the head $m_{p/2} = (Out, In)$, we also have only one traceable binding. The resulting bound traceable credential is:

$$(8)\ \ p(a, X)^{(Out,In)} \longleftarrow r(Y, X)^{(Out,In)}, q(b, Y)^{(Out,In)}.$$

Summarising, *BoundSet*(cl) consists of eight bound traceable credentials. From these, at least one bound traceable credential from credentials (1)-(6), credential (7), and credential (8) must be physically deployed. The user (or the system integrator) may also choose to store all credentials from the bound set in order to improve the reliability of the credential discovery. According to Definition 6.4.5, the depositary of credentials (1) to (7) is a, and the depositary of credential (8) is b.

How the mode information is embedded into a logical clause is implementation dependent. For example, the mode information may be encoded as an additional ground argument of a credential. In Standard TuLiP (Chapter 5) we take a different approach, and we rewrite the name of a predicate symbol, so that the predicate symbol name already encodes the associated mode. For instance, given credential *student*(*ut*, *alice*, 23) and $\mathcal{M}$(*student*/3) = $\{(In, In, Out), (In, Out, Out), (Out, In, Out)\}$, we write:

$$(1) : \textit{student\_ioo}(\textit{ut}, \textit{alice}, 23).$$
$$(2) : \textit{student\_iio}(\textit{ut}, \textit{alice}, 23).$$
$$(3) : \textit{student\_oio}(\textit{ut}, \textit{alice}, 23).$$

Here suffix *ioo* states for (In, Out, Out), *iio* states for (In, In, Out), and *oio* states for (Out, In, Out).

Redundant Storage and Constraints A constraint can have only one mode. This is because the depositary of a defining constraint clause is the same as the depositary of the credential that contains the constraint (see Definition 6.3.2). Therefore redundancy in the storage of a constraint clause does not come from the multiple modes assigned to a constraint but from the redundancy in the storage of a credential containing this constraint. If a credential has multiple depositaries, so has each constraint clause related to this credential.

6.5 LIAR

In this section we present the extended version of LIAR - Lookup and Inference AlgoRithm. We show how LIAR gives answer to a query and how LIAR fetches the remote clauses needed for the query evaluation.

LIAR operates on a state (first introduced in Chapter 4). Because now the state contains also user-defined constraints and also because now we use packages rather than raw credentials, we need to update the original definition of a state.

Definition 6.5.1 *A* state $\mathcal{P}$ *is a finite collection of pairs* (a, P_a) *where* P_a *is a collection of packages and* a *is the depositary of these packages. A state also includes a set of system-wide built-in constraint clauses* C*.*

The extended LIAR algorithm is structured in a similar way as the original LIAR algorithm we present in Chapter 4. The most important difference is the evaluation of the constraints and the use of packages. The input to the algorithm is a package containing a bound query, the constraint clauses defining the constraints in the query and the evaluation algorithm for the constraints in the query. The output of the algorithm is the FACTSTACK containing the ground answers to the query.

In what follows, we define LIAR and present the pseudo code. Next, we give a more detailed description of how the algorithm works and give an example. Finally, we discuss the declarative semantics of the extended state and the soundness and completeness of the extended algorithm.

Before we proceed we need to update Definition 4.4.1 to take bound atoms into account and we need to introduce some auxiliary notation (partly repeated from Chapter 4 for the convenience of the reader).

Definition 6.5.2 (Connected) *We say that two bound atoms* A^{m_A} *and* B^{m_B} *are* connected *if the following two conditions are simultaneously satisfied:*

- $m_A(\text{issuer}) = m_B(\text{issuer}) = \mathit{Out}$ *and* $m_A(\text{subject}) = m_B(\text{subject}) = \mathit{In}$*,*
- $\text{subject}(A)$ *is ground and* $\text{subject}(A) = \text{subject}(B)$*.*

Let A be an atom and S be a set of atoms. We adopt the following conventions:

(i) We write $A \mathrel{\widetilde{\in}} S$ iff $\exists A' \in S$, such that $A' \approx A$ (i.e. A' is a renaming of A).

(ii) We write $A \mathrel{\widetilde{\notin}} S$ iff $\nexists A' \in S$ such that $A' \approx A$.

(iii) We write $A \overset{\theta}{\hookrightarrow} S$ iff $\exists A' \mathrel{\widetilde{\in}} S$ standardised apart w.r.t. A such that $\gamma = mgu(A, A')$ and $A\theta \approx A\gamma$.

(iv) We write $A \mathrel{\widetilde{\notin}_m} S$ if $A \mathrel{\widetilde{\notin}} S$ or $\forall A' \in S$, such that $A' \approx A$, $m_{A'} \neq m_A$.

Definition 6.5.3 *Let* $\mathbf{A} : \zeta$ *be a package where* $\mathbf{A} = A_1^{m_{A_1}}, \ldots, A_n^{m_{A_n}}$ *is a well-moded bound query and* ζ *the set of the related constraints and the constraint evaluation algorithm. We define the Lookup and Inference AlgoRithm (LIAR) which given a state* $\mathcal{P}$ *and package* $\mathbf{A} : \zeta$ *as the input returns the (possibly empty) sets of atoms* FACTSTACK *and* GOALSTACK*. The algorithm is shown in Listing 6.1.*

Remark 6.5.4 *LIAR operates on bound atoms and credentials. Therefore, in Listing 6.1 all atoms and credentials are bound. For sake of clarity, however, we skip the superscripts carrying the mode information. We use the following rule: if* A *is a bound atom, then its mode is* m_A*.*

INPUT: $\mathbf{A} : \zeta_A$. /* $\mathbf{A}$ is the initial query and
ζ_A the corresponding solver package. */
Init :
CLSTACK : $\{\Box \longleftarrow \mathbf{A} : \zeta_A\}$;
FACTSTACK = GOALSTACK = CONSTRAINTSTACK = VISITED = $\emptyset$;
SATISFIED = **FALSE** ;
REPEAT
Phase 1 (Top-down resolution):
CHOOSE:
$c : \ H \longleftarrow \mathbf{B}, C, \mathbf{D} : \zeta \in$ CLSTACK and
$\mathbf{B}' \subseteq$ FACTSTACK, such that the following conditions hold:
(i) $\mathbf{B}$ and $\mathbf{B}'$ unify with mgu θ,
(ii) $C\theta$ is well-moded,
(iii)**IF** C is a credential atom **THEN**
$C\theta \overset{\sim}{\notin}_m$ GOALSTACK;
IF $m_C(issuer) = Out$ and $m_C(subject) = In$ **THEN**
subject$(C\theta) \notin$ VISITED ;
ENDIF
ENDIF
IF C is a constraint **THEN**
IF $\zeta(C\theta)$ succeeds with c.a.s. $\gamma_1, \ldots, \gamma_n$**THEN**
FOR EACH $\gamma_i \in \{\gamma_1, \ldots, \gamma_n\}$ **DO**
IF $C\theta\gamma_i \notin$ FACTSTACK **THEN**
ADD $C\theta\gamma_i$ to FACTSTACK;
ENDIF
END FOR EACH
IF nothing has been added to FACTSTACK**THEN**
goto *Phase 1*;
ELSE
goto *Phase 2*;
ENDIF
ELSE
goto *Phase 1*;
ENDIF
ELSE
/ C is a credential atom */*
Let $\mathbf{E}$ be the maximum prefix of $\mathbf{D}$ such that
$\forall E \in \mathbf{E}$, E is a constraint atom;
IF $\mathbf{E}$ is empty **THEN**
ADD $(C\theta, \mathit{true} : \zeta)$ to CONSTRAINTSTACK;
ELSE
ADD $(C\theta, \mathbf{E}\theta : \zeta)$ to CONSTRAINTSTACK;
ENDIF
ENDIF
ADD $C\theta$ to GOALSTACK;
IF $m_C(issuer) = In$ **THEN**
FETCH at *issuer*$(C\theta)$ all packages $\{c_1 : \zeta_1, \ldots, c_n : \zeta_n\}$ such that
$\forall i, i \in [1, n], m_{head(c_i)} = m_C$ and $head(c_i)$ unifies
with $C\theta$ with mgu γ_i ;
FOR EACH $c_i\gamma_i : \zeta_i \in \{c_1\gamma_1 : \zeta_1, \ldots, c_n\gamma_n : \zeta_n\}$**DO**

```
            IF c_i γ_i : ζ_i ∉~ CLSTACK THEN ADD c_i γ_i : ζ_i to CLSTACK ENDIF
52      END FOR EACH
    ELSEIF m_C(issuer) = Out and m_C(subject) = In THEN
54      FETCH all packages {c_1 : ζ_1, ..., c_n : ζ_n} stored at subject(Cθ) such that
            ∀i, i ∈ [1,n], m_head(c_i)(issuer) = Out and m_head(c_i)(subject) = In ;
56      ADD subject(Cθ) to VISITED;
        FOR EACH c_i : ζ_i ∈ {c_1 : ζ_1, ..., c_n : ζ_n} DO
58          IF c_i : ζ_i ∉~ CLSTACK THEN ADD c_i : ζ_i to CLSTACK ENDIF
        END FOR EACH
60  ENDIF
    Phase 2 (Bottom-up model-building):
62  REPEAT
        CHOOSE: H ⟵ B ∈ CLSTACK and B' ⊆ FACTSTACK,
64          such that B and B' unify with mgu θ ;
        IF Hθ ∉ FACTSTACK THEN
66          IF ∃(H', E : ζ) ∈ CONSTRAINTSTACK such that
                H' and Hθ unify with mgu γ and
68              ζ(Eγ) succeeds THEN
                    ADD Hθ to FACTSTACK;
70          ENDIF
        ENDIF;
72      IF m_H(issuer) = Out AND m_H(subject) = In
            AND issuer(Hθ) ∉ VISITED THEN
74          ADD to CLSTACK the package:
                dummy(X, issuer(Hθ))^(Out,In) ⟵
76                          dummy(X, issuer(Hθ))^(Out,In). : □
                where □ is a dummy empty constraint solver.
78      ENDIF
    UNTIL nothing can be added to FACTSTACK;
80  IF A is ground and A ⊆ FACTSTACK THEN SATISFIED = TRUE ENDIF
UNTIL SATISFIED OR nothing can be added to FACTSTACK and CLSTACK;
82 OUTPUT = FACTSTACK, GOALSTACK;
```

Listing 6.1: The Lookup and Inference AlgoRithm (LIAR)

The algorithm maintains four *stacks*: CLSTACK contains the set of packages collected so far, FACTSTACK contains the set of atomic logical consequences inferred from CLSTACK, and GOALSTACK contains the set of bound atomic goals already processed (to handle loops). The CONSTRAINTSTACK contains pairs of the form $(\mathit{Goal}, \mathit{Constraint})$ where *Goal* is a bound atom and *Constraint* is either a conjunction of bound atoms, or a constant *true*. Additionally, the VISITED stack contains the set of entities that have been visited during the processing of subject traceable chains.

Initially, CLSTACK contains a single package corresponding to the initial query $\mathbf{A}$; the other stacks are empty. The algorithm is divided in two phases. Phase 1 performs the credential (package to be precise) discovery and the constraint evaluation. First, it selects a *new* well-moded (bound) atom $C\theta$ from the body of a (bound) credential from a package in CLSTACK such that there exists a prefix $\mathbf{B}$ in the body which unifies with some $\mathbf{B}' \in$ FACTSTACK. If $\mathbf{B}$ is empty, then the first atom from the body is selected as the new goal. If given the selected goal $C\theta$, C is a credential atom, the algorithm checks the mode

m_c of C. If $m_C(\textit{issuer}) = \textit{Out}$ and $m_C(\textit{subject}) = \textit{In}$ then it checks if the $\textit{subject}(C\theta)$ has not been visited yet. In such a case there is no need to add this atom to the GOALSTACK because all the related subject traceable clauses has been already fetched. After selecting a new goal, the algorithm checks whether it is a constraint or a credential atom.

If the selected goal is a constrain, the algorithm evaluates the constraint using the algorithm provided in the package corresponding to this constraint. If a package does not contain an evaluation algorithm or the constraint is a built-in constraint then the default Prolog engine is used to evaluate the constraint. The effect of evaluating a constraint may be one or more (constraint) facts. Each such fact is added to FACTSTACK (if not already there), and then the algorithm proceeds to Phase 2. If the result of the evaluation is an empty set of computed answer substitutions or nothing was added to FACTSTACK, the algorithm returns to Phase 1 and tries to select another goal. If the evaluation yields at least one computed answer, the algorithm proceeds to Phase 2.

If C is a credential atom, the pair $(C\theta, \mathbf{E}\theta)$ is added to the CONSTRAINTSTACK in order to record the constraints on which $C\theta$ may depend. Here, in order to simplify the algorithm, we decided to look only at the set of constraint atoms E directly following C in the clause. Later, when the algorithm generates a new fact H, before adding it to the FACTSTACK, it can check if there exists at least one pair $(H'\theta, \mathbf{E}) \in$ CONSTRAINTSTACK such that H' unifies with H with mgu γ, and $\mathbf{E}\gamma$ is satisfied. If such a pair is not in the CONSTRAINTSTACK then H does not contribute to the solution set and should be discarded (though it is possible that H will be accepted later after new constraints are added to CONSTRAINTSTACK on which H depends and at least one of them is satisfied). Recall that when selected goal is a credential atom and the constraint prefix $\mathbf{E}$ (lines 36-43) is empty, the pair $(C\theta, \textit{true})$ is added to the CONSTRAINTSTACK in Phase 1 meaning that every newly generated fact H such that H and $C\theta$ unify should be added to the FACTSTACK regardless of other existing constraints.

Having selected a new goal which is a credential atom, the algorithm fetches the *new* packages from either $\textit{issuer}(C\theta)$ or $\textit{subject}(C\theta)$. If mode of the selected new goal C, m_C, is such that $m_C(\textit{issuer}) = \textit{In}$ then LIAR fetches all the packages $c : \zeta$ from $\textit{issuer}(C\theta)$ in which $head(c)$ has mode m_C and unifies with $C\theta$. When mode of C, m_C, is such that $m_C(\textit{issuer}) = \textit{Out}$ and $m_C(\textit{subject}) = \textit{In}$ then *all* packages $c : \zeta$ such that $m_{head(c)}(\textit{issuer}) = \textit{Out}$ and $m_{head(c)}(\textit{subject}) = \textit{In}$ are fetched from $\textit{subject}(C\theta)$ and added to CLSTACK.

In Phase 2, the model of the set of clauses in the CLSTACK is build bottom-up. Newly inferred facts are added to the FACTSTACK (after checking the constraints). For a fact having mode m such that $m(\textit{issuer}) = \textit{Out}$ and $m(\textit{subject}) = \textit{In}$, the algorithm adds a *dummy* clause to CLSTACK, so that the subject traceable chains can be discovered properly (see also the description of the original LIAR algorithm in Chapter 4).

In the following example we demonstrate the evaluation of an example query on the policy presented in Example 6.2. Here we concentrate on the constraints. We present the credential discovery in Chapter 4.

Example 6.8 We assume the set of clauses from Example 6.2 and the mode assignment as in Example 6.3. Therefore, we have the state in which *comp* stores the following two packages:

1. $\textit{Pkg}_1 = \textit{cl}_1 : \zeta_1$:

- cl_1:

  (1) *employee*$(comp, X, Y)$ $\longleftarrow$ *employee_int*$(comp, X, Y2)$, *filter*$(Y2, Y)$.

- ζ_1:

  (2) *filter*$([\,], [\,])$.
  (3) *filter*$([AttName : AttValue|Atts], Atts2)$ $\longleftarrow$
  confidential$(AttName)$, *filter*$(Atts, Atts2)$.
  (4) *filter*$([AttName : AttValue|Atts], [AttName : AttValue|Atts2])$ $\longleftarrow$
  non-confidential$(AttName)$, *filter*$(Atts, Atts2)$.
  (5) *confidential*$(salary)$.
  (6) *confidential*$(bankaccount)$.
  (7) *non-confidential*$(position)$.

2. $Pkg_2 = cl_2 : \zeta_2$:

   - cl_2:

     (8) *employee_int*$(comp, marcin,$
     [salary:2500,position:phd,bankaccount:123456]$)$.

   - $\zeta_2 = \emptyset$.

We assume that the following query is issued:

$Q =$ *employee*$(comp, marcin, X) : \emptyset$.

The query does not contain any constraints: therefore the ζ component of the query is the emptyset. Below we show the contents of CLSTACK, GOALSTACK, CONSTRAINTSTACK, and FACTSTACK as the algorithm progresses.

After initialisation the contents of the stacks is the following:

CLSTACK :
$(P_1) : \square \longleftarrow$ *employee*$(comp, marcin, X) : \emptyset$.
GOALSTACK : $\emptyset$.
CONSTRAINTSTACK : $\emptyset$.
FACTSTACK : $\emptyset$.

LIAR selects $C\theta =$ *employee*$(comp, marcin, X)$ as the new goal (lines 9-19). The new goal is a credential atom. Because $C\theta$ is not followed by any constraint, the following element is added to CONSTRAINTSTACK (lines 36-43):

(*employee*$(comp, marcin, X)$, *true* : $\emptyset$).

The mode associated with *employee*$/3$ is (In, In, Out). Therefore, LIAR knows that it should fetch the related packages from *comp*. *comp* stores two packages: Pkg_1 and Pkg_2. Pkg_1 contains a credential whose head unifies with the selected new goal. LIAR fetches

Pkg_1 and the instance of Pkg_1 is added to CLSTACK (lines 46-52). Because FACTSTACK is empty, the algorithm returns to Phase 1 with the following contents of the CLSTACK, GOALSTACK, CONSTRAINTSTACK, and FACTSTACK:

CLSTACK :

(P_1) : $\square \longleftarrow$ *employee*$(comp, marcin, X) : \emptyset$.

(P_2) : *employee*$(comp, marcin, Y) \longleftarrow$ *employee_int*$(comp, marcin, Y2)$, *filter*$(Y2, Y)$. : ζ_1

GOALSTACK :

(G_1) : *employee*$(comp, marcin, X)$

CONSTRAINTSTACK :

(C_1) : (*employee*$(comp, marcin, X)$, *true* : $\emptyset$)

FACTSTACK : $\emptyset$.

The algorithm selects $C\theta =$ *employee_int*$(comp, marcin, Y2)$ to be the new goal. C is a credential atom, but now C is followed in the containing clause (cl_1) by constraint *filter*$(Y2, Y)$. Therefore, CONSTRAINTSTACK is extended with the following element (lines 36-43):

(C_2) : (*employee_int*$(comp, marcin, Y2)$, *filter*$(Y2, Y) : \zeta_1$).

Because *mode*(*employee_int*) $= (In, In, Out)$, LIAR knows that the related packages must be stored by *comp*. Package Pkg_2 contains a credential whose head unifies with the selected goal. Pkg_2 is fetched and the instance of Pkg_2 is added to CLSTACK (lines 46-52). The algorithm proceeds to Phase 2. Here, at some point, *Pkg_2* is selected from CLSTACK which yields:

$H\theta =$ *employee_int*$(comp, marcin,$ *[salary:2500,position:phd,bankaccount:123456]*$)$.

Before adding $H\theta$ to FACTSTACK, LIAR first checks is there exists at least one related constraint in CONSTRAINTSTACK. In this case, the constraint on which $H\theta$ depends is *filter*$(Y2, Y)$ and the corresponding element in CONSTRAINTSTACK is C_2. Therefore, we have (lines 65-70):

$\mathbf{E} =$ *filter*$(Y2, Y)$

$\zeta = \zeta_1$

$\gamma = \{$*Y2/[salary:2500,position:phd,bankaccount:123456]*$\}$.

Because ζ_1(*filter*(*[salary:2500,position:phd,bankaccount:123456]*$, Y)$) succeeds, the following fact ($H\theta$) is added to FACTSTACK:

employee_int$(comp, marcin,$ *[salary:2500,position:phd,bankaccount:123456]*$)$.

When returning to Phase 1, the contents of the CLSTACK, GOALSTACK, CONSTRAINTSTACK, and FACTSTACK is therefore the following:

CLSTACK :

(P_1) : $\square \longleftarrow$ *employee*$(comp, marcin, X) : \emptyset$.

(P_2) : *employee*$(comp, marcin, Y) \longleftarrow$ *employee_int*$(comp, marcin, Y2)$,

$\mathit{filter}(\mathit{Y2}, Y).:\zeta_1$

$(P_3):$ $\mathit{employee\_int}(\mathit{comp}, \mathit{marcin},$
$\mathit{[salary:2500,position:phd,bankaccount:123456]}).:\emptyset$

GOALSTACK :

$(G_1):$ $\mathit{employee}(\mathit{comp}, \mathit{marcin}, X)$

$(G_2):$ $\mathit{employee\_int}(\mathit{comp}, marcin, \mathit{Y2})$

CONSTRAINTSTACK :

$(C_1):$ $(\mathit{employee}(\mathit{comp}, \mathit{marcin}, X), \mathit{true}:\emptyset)$

$(C_2):$ $(\mathit{employee\_int}(\mathit{comp}, \mathit{marcin}, \mathit{Y2}), \mathit{filter}(\mathit{Y2}, Y):\zeta_1)$

FACTSTACK :

$(F_1):$ $\mathit{employee\_int}(\mathit{comp}, \mathit{marcin},$
$\mathit{[salary:2500,position:phd,bankaccount:123456]})$

Back in Phase 1, LIAR chooses package P_2 for which we have:

$\mathbf{B} = \mathit{employee\_int}(\mathit{comp}, marcin, \mathit{Y2})$
$\mathbf{B}' = \mathit{employee\_int}(\mathit{comp}, \mathit{marcin}, \mathit{[salary:2500,position:phd,bankaccount:123456]})$
$\theta = \{\mathit{Y2}/\mathit{[salary:2500,position:phd,bankaccount:123456]}\}.$

Thus, the new goal $C\theta$ is:

$C\theta = \mathit{filter}(\mathit{Y2}, Y)\theta = \mathit{filter}(\mathit{[salary:2500,position:phd,bankaccount:123456]}, Y).$

The newly selected goal is a user-defined constraint. Therefore, $C\theta$ is evaluated using the information (ζ_1) provided in the package ($P_2 = \mathit{Pkg}_1$) corresponding to the containing clause (cl_1). The ζ_1 component of Pkg_1 contains a well-moded program defining the constraint $\mathit{filter}/2$. Therefore, $C\theta$ will be evaluated using the LD-resolution on query $C\theta$ and program ζ_1, i.e. $\zeta_1(C\theta)$. The result of this evaluation is the computed answer substitution (line 21):

$\gamma = \{Y/\mathit{[position:phd]}\}.$

As the result, $C\theta\gamma = \mathit{filter}(\mathit{[salary:2500,position:phd,bankaccount:123456]}, \mathit{[position:phd]})$ is added to FACTSTACK (lines 22-26).
Next, the algorithm moves to Phase 2. Here, at some point, LIAR chooses package (line 64):

$(P_2):$ $\mathit{employee}(\mathit{comp}, \mathit{marcin}, Y) \longleftarrow \mathit{employee\_int}(\mathit{comp}, marcin, \mathit{Y2}),$
$\mathit{filter}(\mathit{Y2}, Y).:\zeta_1.$

We have:

$\mathbf{B} = \mathit{employee\_int}(\mathit{comp}, \mathit{marcin}, \mathit{Y2}), \mathit{filter}(\mathit{Y2}, Y)$
$\mathbf{B}' = \mathit{employee\_int}(\mathit{comp}, \mathit{marcin}, \mathit{[salary:2500,position:phd,bankaccount:123456]}),$
$\mathit{filter}(\mathit{[salary:2500,position:phd,bankaccount:123456]}, \mathit{[position:phd]})$
$\theta = \{Y/\mathit{[position:phd]}, \mathit{Y2}/\mathit{[salary:2500,position:phd,bankaccount:123456]}\}$
$H\theta = \mathit{employee}(\mathit{comp}, \mathit{marcin}, \mathit{[position:phd]}).$

Before $H\theta$ can be added to FACTSTACK, LIAR again checks CONSTRAINTSTACK. In this case, the only related element in CONSTRAINTSTACK is C_1. Because for C_1, $\zeta = \mathit{true}$, this means that $H\theta$ does not depend on any constraint and therefore can be added to FACTSTACK.

Because no new facts can be generated, the algorithm terminates returning one answer: $\mathit{employee}(\mathit{comp}, \mathit{marcin}, \mathit{[position:phd]})$.

6.5.1 Declarative Semantics, Soundness and Completeness

We show the declarative semantics for the original state (without user-defined constraints) in Chapter 4 and we also prove that LIAR (again, without user-defined constraints) is sound and complete with respect to the declarative semantics. In Core TuLiP, we only have built-in constraints. In TuLiP, a user may provide her own constraint evaluation algorithm. This makes the declarative semantics harder to define because the constraint evaluation algorithm is - in general - unknown. The declarative semantics of a constraint evaluation algorithm can be given if the evaluation algorithm is given in terms of a well-moded Prolog program (see the comment under Definition 6.3.3) and the LD-resolution. In this case the definition of each constraint in a package is given by one or more constraint clauses. Because the constraints and the defining constraint clauses are user-defined, different users can use the same predicate names but mean different things. We call this problem "name clashing". In what follows we assume that the name clashing is avoided (e.g. by renaming). Under these assumptions, the declarative semantics of a state is given in terms of logic programming as follows:

Definition 6.5.5 *Let $\mathcal{P}$ be the state $\{(a_1, P_1), \ldots, (a_n, P_n), C\}$, and A be an atom.*

- *We denote by $P(\mathcal{P})$ the set of (credential or constraint) clauses $C_1 \cup \cdots \cup C_n \cup C$ where C_i, $i \in [1, n]$ is the set of clauses contained in the corresponding set of packages P_i. We call $P(\mathcal{P})$ the LP-counterpart of state $\mathcal{P}$.*
- *We say that A is true in state $\mathcal{P}$ iff $P(\mathcal{P}) \models A$.*

Bellow we present the updated proofs of soundness and completeness of the extended LIAR algorithm. We start with the updated version of Lemma 4.4.3.

Lemma 6.5.6 *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of the algorithm execution for some well-moded query. Let A be an atom in* FACTSTACK. *Then A is ground.*

Proof. The proof proceeds by induction on the size of FACTSTACK. In the basic case FACTSTACK is empty and so the proposition holds for any atom in the FACTSTACK.
Now, assume that FACTSTACK contains only ground atoms. We prove that each time a new atom is added to FACTSTACK, this atom is ground. An atom is added to FACTSTACK in two cases:

1. as the result of the constraint evaluation in Phase 1 of the algorithm, or

2. as the result of the bottom-up evaluation of the facts in FACTSTACK and a credential clause selected from CLSTACK in Phase 2 of the algorithm.

Case 1: The atom is a constraint and is added to FACTSTACK as the result of the constraint evaluation in Phase 1 of the algorithm. Recall that constraints are defined in terms of a well-moded Prolog program.
Let $C\theta$ be the selected goal such that C is a constraint atom. By construction of the algorithm $C\theta$ is well-moded. This means that all the input positions of $C\theta$ are ground. By Definition 6.3.3, the evaluation algorithm respects the modes, and therefore for any computed answer substitution σ resulting from the evaluation of the well-moded constraint $C\theta$, we have that $C\theta\sigma$ is ground and will be added to FACTSTACK.

Case 2: The atom is a credential atom added to the FACTSTACK as the result of the bottom-up evaluation of the facts in FACTSTACK and a credential clause selected from CLSTACK in Phase 2 of the algorithm. But then, by Lemma 4.4.3, any new credential atom added to FACTSTACK is ground. □

Now we can prove the soundness and the completeness of the extended LIAR algorithm w.r.t. the declarative semantics.

Theorem 6.5.7 (soundness) *Let $\mathcal{P}$ be a state and* FACTSTACK *be the result of executing the extended LIAR on $\mathcal{P}$ and a well-moded query. Then $\forall A \in$ FACTSTACK, $P(\mathcal{P}) \models A$.*

Proof. We have two cases:

1. A is a constraint atom.
2. A is a credential atom.

If A is a constraint atom then by the assumption that the constraint evaluation algorithm is given in terms of a Prolog program and the SLD resolution the thesis follows from the soundness of the SLD resolution. If A is a credential atom then by Theorem 4.4.4 $P(\mathcal{P}) \models A$. □

Theorem 6.5.8 (completeness) *Let $\mathcal{P}$ be a state and then* FACTSTACK, GOALSTACK *be the result of executing the extended LIAR on $\mathcal{P}$ and a given well-moded goal.*
Then $\forall C \in$ GOALSTACK, if $\exists$ a successful SLD derivation $\delta : C \xrightarrow{\theta}_{P(\mathcal{P})} \square$ then $C \overset{\theta}{\hookrightarrow}$ FACTSTACK.

Proof. First we prove that the theorem holds for any constraint in GOALSTACK.
If $C \in$ GOALSTACK and C is a constraint atom then, by the construction of the algorithm, C is evaluated using either built-in or user-defined algorithm (fetched with the package containing the credential with this constraint atom). Since we are assuming that the constraint evaluation algorithm is given by a set of well-moded clauses and the SLD-resolution the thesis follows by definition.
Now, let C be a (bound) credential atom in GOALSTACK. Then, by Proposition 4.4.6 and by the fact that the theorem holds for any constraint in GOALSTACK, $C \overset{\theta}{\hookrightarrow}$ FACTSTACK. □

6.6 Prolog Markup Language (PML)

TuLiP supports positional arguments in a natural way. This, however, requires additional knowledge when interpreting a credential or a constraint atom as the intended meaning of an

argument is not explicit. Consider for instance the following credential atom:

$$\text{student}(\text{ut}, \text{alice}, 23, \text{cs}, \text{tel}, 0123456)$$

Whereas the meaning of the first two arguments is clear from the definition of a credential atom (the issuer the subject) and, what is the intended meaning of the remaining arguments? Prior using this credential atom in a credential, the credential issuer must know the precise meaning of each of the arguments. Even though this process may be automated to some extent, referring to an argument by the argument position may be fragile and error prone. Additionally, further argument parsing, matching between semantically similar arguments, and other argument related transformations are hard to automate without additional knowledge which may often require human interaction. The situation may be improved if an argument is named, i.e. the argument can be uniquely identified by the name without the need of referring to the position of an argument within the credential or constraint atom. One possible way of handling named attributes is to use XML [97]. In XML, we can easily assign names to attributes (as the names of XML elements and attributes). Additionally, as XML content is also ordered (in the so called *document order*), the structure of the XML data set can be used as the additional information source whenever applicable. For instance, the intended meaning of the arguments in the credential atom above can be clarified by using the following XML structure:

```
<student  id="0123456">
    <age>23</age>
    <department>cs</department>
    <study>tel </study>
</student >
```

XML is a popular document interchange format and is also frequently used as a practical language for policy exchange [21, 77] (see Chapter 5, where we also use XML to encode a Standard TuLiP credential).

In order to support named attributes in TuLiP we developed the so called Prolog Markup Language (PML) which facilitates the use of the structured XML content in a credential or in a constraint atom. PML encodes the XML data using the so called *field-notation* [90]. The field-notation is based on the observation that, syntactically, both XML and Prolog are based on nested structures (nested elements in XML and function symbols in Prolog). By using PML, one can easily use XML structured data in credential and constraint atoms:

$$\text{student}(\text{ut}, \text{alice}, \text{student:[id:0123456]:[age:23,department:cs,study:int]}).$$

Here, the third argument is a so called *PML-Term*. Below, we define PML formally. In particular, we show how XML content can be transformed to the PML equivalent and vice versa.

Definition 6.6.1 *We say that the Prolog term $T : As : C$ is a* PML-Term *if the following conditions hold:*

- *T is an atomic Prolog ground term,*
- *As is a (possible empty) ground Prolog list of the form $[a_1 : v_1, \ldots, a_n : v_n]$ where for each $i \in [1, n]$, a_i, v_i are terms,*

- *C is an atomic Prolog ground term of arity zero or a list $[c_1, \ldots, c_n]$ where for each $i \in [1, n]$, c_i is another PML-Term.*

We call T the tag*, C the* content*, and $As = [a_1 : v_1, \ldots, a_n : v_n]$ the* attribute list*, where each a_i is called an attribute and v_i is the corresponding value. If the attribute list is empty we write $T : C$ instead of $T : [] : C$.*

Now we have to show how an XML element can be mapped to a PML term and vice versa. We assume that each XML element has the following form:

$$el = \text{<}T\ a_1 = \text{“}v_1\text{”}, \ldots, a_n = \text{“}v_n\text{”>}C\text{</}T\text{>}$$

We call el a simple XML element. Here T is the name of a simple XML element, $a_i, i \in [1, n]$ is a simple XML attribute, $v_i, i \in [1, n]$ is the value of attribute a_i, and C is either text without any markup or another simple XML element.

Let L be a first order language, and let U_L be the Herbrand universe for L. Let $Strings$ be the set of all possible (XML) strings of character data. We assume that there exists an injection $\tau : Strings \rightarrow U_L$ which assigns a Prolog ground term to every strings of character data. Similarly, we define $\varsigma : U_L \rightarrow Strings$ such that $\tau \circ \varsigma = id_{Strings}$. Thus, for any s, t such that $s \in Strings, t \in U_L$, if $\tau(s) = t$ then $\varsigma(t) = s$. We denote $\tau(s)$ by s^τ and $\varsigma(t)$ by t^ς. Therefore, for any $s \in Strings, t \in U_L$, we have that $s^{\tau^\varsigma} = s$.

Definition 6.6.2 (XML to PML) *Let el=<e a_1 = “v_1”, …, a_n = “v_n”>CONTENT</e> be a simple XML element. The corresponding PML-Term, $pml(el)$, is defined as follows:*

- *if* CONTENT *is of type* text *then*

$$pml(el) = e^\tau : [a_1^\tau : v_1^\tau, \ldots, a_n^\tau : v_n^\tau] : \text{CONTENT}^\tau;$$

- *otherwise* CONTENT $= el_1 \ldots el_m$*, $m \geq 0$, and*

$$pml(el) = e^\tau : [a_1^\tau : v_1^\tau, \ldots, a_n^\tau : v_n^\tau] : [pml(el_1), \ldots, pml(e_m)].$$

The position of the tags and attributes in a PML-Term is the same as the position of the element tags and attributes in the corresponding XML document.

Given a PML-Term, we construct the corresponding simple XML element as follows:

Definition 6.6.3 (PML to XML) *Let $pl = e : [a_1 : v_1, \ldots, a_n : v_n] :$ CONTENT be a PML-Term. The corresponding (simple) XML element, $xml(pl)$, is constructed as follows:*

- *if* CONTENT *is an atomic ground term then*

$$xml(pl) = \text{<}e^\varsigma\ a_1^\varsigma = \text{“}v_1^\varsigma\text{”}, \ldots, a_n^\varsigma = \text{“}v_n^\varsigma\text{”> CONTENT}^\varsigma \text{</}e^\varsigma\text{>};$$

- *if* CONTENT $= [pl_1, \ldots, pl_m], m > 0$ *then*

$$xml(pl) = \text{<}e^\varsigma\ a_1^\varsigma = \text{“}v_1^\varsigma\text{”}, \ldots, a_n^\varsigma = \text{“}v_n^\varsigma\text{”>}\ xml(pl_1) \ldots xml(pl_m)\ \text{</}e^\varsigma\text{>};$$

- *if* CONTENT $= [\,]$ *then*

$$xml(pl) = <e^{\varsigma}\ a_1^{\varsigma} = \text{“}v_1^{\varsigma}\text{”}, \ldots, a_n^{\varsigma} = \text{“}v_n^{\varsigma}\text{”}></e^{\varsigma}>;$$

- *undefined otherwise.*

By the proper construction of the mappings τ and ς, every simple XML element can be converted to a PML-term. Non-simple elements and also other XML entities, like *XML Document Type Definitions* (DTD), schemas, *XML Prolog*, *processing instructions*, or *comments* [97], cannot be converted to the PML form.

6.7 Related Work

We present the related work on Trust Management in Chapter 2 and the related work on Core TuLiP in Chapter 4. To our best knowledge, no similar approach to handling distributed constraints and redundant credential storage exists. To some extent (non-distributed) constraints can be specified in other trust management languages having logic based semantics. For instance, constraints can be expressed in $\mathcal{X}$-TNL [22] or to some extent in Delegation Logic [64]. Also, constraints can be used in XACML [77] in the form of obligations.

6.8 Conclusions

In this chapter we define TuLiP, a trust management system based on Core TuLiP (see Chapter 4). While Core TuLiP gives us strong fundaments, TuLiP provides the necessary extensions and formalisms that make Core TuLiP deployable and suitable for practical use. In particular, in TuLiP we formalise the notion of redundant credential storage introduced in Chapter 5 where we present Standard TuLiP. TuLiP also extends Standard TuLiP in allowing an unlimited number of arguments in a credential atom, supporting external constraint solvers, and facilitating the use of named attributes by introducing the Prolog Markup Language.

The concept of redundant storage is formalised by introducing the notion of bindings and bound credentials, which draw a clear separation between theory and practice, between the pure declarative meaning of a credential and how the credential is actually deployed and used in a real system. Bound credentials are also the first class objects for our extended Lookup and Inference AlgoRithm (LIAR).

In TuLiP we also formally define a constraint. In particular, we introduce a user-defined constraint and we show how a user-defined constraint can be evaluated by introducing the notion of a package containing the credential, all related used-defined constraint clauses and the user-defined algorithm for evaluating the constraints occurring in the credential.

We prove that the extended LIAR is sound and complete with respect to the standard theoretical logic programming semantics.

Future work The expressive power of TuLiP can be further extended by admitting negation in the language. We investigate non-monotonic polices in Chapter 3 where we extend the RT family with the so called negation in context. Before adding non-monotic features

to TuLiP, we also need to investigate what consequences this extension has on the storage type system. Finally, we would like to incorporate into TuLiP the grouping and aggregate operations which we describe in Chapter 7.

CHAPTER 7

LP with Flexible Grouping and Aggregates Using Modes

As we say in the Introduction (Chapter 1), a flexible trust management language should be prepared to accommodate extensions allowing the user to model not only credential based but also reputation based scenarios.

The role of a reputation system is to collect, distribute, and aggregate feedbacks concerning past behaviour of the users. Each such a feedback is usually in the form of a trust metric which is often from a domain which can be either infinite (continuous) or finite (multivalued). Feedback is collected from a set of entities (often called recommenders) and then aggregated, for example in order to compute the average trust level for the seller. For an aggregate operation to be useful it is often crucial to be able to select a subset of all the feedbacks we are able to collect. For instance, one usually wants to compute average feedback for a concrete user acting in a specific role, not for all the users and for all the roles the users play in the system. It is then crucial to *group* the feedback according to some criteria.

TuLiP can deal with collecting and distributing feedback by using credential arguments. However, TuLiP does not have grouping. In other words TuLiP does not allow us to collect all instances of a query and group the results by one or more query argument.

Therefore, solving the problem of grouping is the first step we need to take if we want to bridge credential-based and reputation-based trust management. Before we can do this in TuLiP however, we must first solve the grouping problem in Logic Programming.

In this chapter we show how to perform grouping in Prolog using the well-known built-in predicate *bagof*. To be included in TuLiP, we have to extend *bagof* with modes, and we have to prove properties regarding groundness of computed answer substitutions and termination which are crucial in trust management.

7.1 Introduction

In a system designed to answer queries (be it a database or a logic program) an aggregate function is designed to be carried out on the set of answers to a given query rather than on a single answer. For example, in a Datalog program containing one entry per employee, one needs aggregate functions to compute data such as the average age or salary of the employee, the number of employees etc.

Grouping and aggregation are useful in practice, and paramount in database systems. In reputation systems, the trust level is computed by aggregating over the feedbacks coming from a specific subset of users. Therefore, if we want TuLiP to be able to express statements such as "employee X will be granted access to confidential document Y provided that the majority of senior executives recommends him", we need to extend TuLiP with grouping and aggregation.

Before we can introduce grouping and aggregation to TuLiP, we first have to investigate how to implement grouping and aggregation in logic programming.

We could choose two possible approaches. In the first approach, grouping and aggregation is implemented as one atomic operation. Here, the result of the grouping operation (a multiset in general) is not visible from the level of the syntax of the language. An advantage of this approach is that multisets do not have to be introduced to the language as first-class citizens. This simplifies the definition of the declarative semantics for the aggregate operations. On the other hand, the grouping data are interesting on its own, especially in Trust Management where sometimes we need to query a specific subset of entities without performing any aggregate operation on it. Therefore, in the second approach, grouping and aggregation are two separate operations. This approach has several advantages. By separating grouping from aggregation one can use the same data set for different aggregate operations. Also, the performance of a trust management system can be improved as the actual network communication is usually far more costly than the evaluation of the grouping and aggregate operations. Finally, the user has more freedom in defining her own aggregate operations. A disadvantage of having grouping and aggregation as separate operations is that in order to be able to define fully declarative semantics for grouping, one needs to extend the language with set-based primitives like *set membership* ($\in$) or set-equation ($=$). This is not trivial task and significant work in this area has been carried out (see Section Related Work). Alternatively, one can use a more practical approach and use a list as a representation of a multiset. Because a list is not a multiset (two lists with different order of the elements are two different lists), the declarative semantics cannot be precise in this case. The Prolog built-in *bagof* is an example of a grouping predicate that uses a list to represent a multiset.

We decided to follow the second approach where grouping and aggregation are two separate operations. We found out that the the built-in *bagof* predicate is expressive enough to be used as a grouping predicate for the trust management language of TuLiP. However, because in TuLiP modes play a crucial role in the storage type system and they guarantee groundness of answers, we have to add modes to the *bagof* predicate, which is not trivial. By having a moded version of *bagof*, we can prove the important properties of the groundness of the computed answer substitutions also for the programs containing grouping operations.

The chapter is structured as follows. In Section 7.2 we present the notational conventions used in this chapter. In Section 7.3 we show how to do grouping in Prolog programs that do not contain grouping subgoals and we show operational semantics by defining the computed

answer substitutions for the grouping goal. In Section 7.4 we show how to use grouping in programs containing grouping goals. Here we generalise the notion of well-moded logic programs to those including grouping subgoals. In Section 7.5 we discuss the properties of the well-moded programs containing grouping atoms. The chapter finishes with Related Work in Section 7.6 and Conclusions in Section 7.7.

7.2 Preliminaries

The preliminaries on Logic Programs are presented in Chapter 4, Section 4.2. Here, we study definite logic programs executed by means of LD-resolution, which consists of the SLD-resolution combined with the leftmost selection rule. A *multiset* is a collection of elements that are not necessarily distinct [73]. The number of occurrences of an element x in a multiset M is its *multiplicity* in the multiset, and is denoted by $mult(x, M)$. When describing multisets we use the notation that is similar to that of the sets, but instead of $\{$ and $\}$ we use $[\![$ and $]\!]$ respectively. For example, $M = [\![1, 1, 2]\!]$ is a multiset where $mult(1, M) = 2$ and $mult(2, M) = 1$.

7.3 Grouping in Prolog

Prolog already provides some grouping facilities in terms of the built-in predicate *bagof*. The *bagof* predicate has the following form:

$$bagof(Term, Goal, List).$$

Term is a prolog term (usually a variable, *Goal* is a callable Prolog goal, and *List* is a variable or a Prolog list. The intuitive meaning of *bagof* is the following: unify *List* with the list (unordered, duplicates retained) of all instances of *Term* such that *Goal* is satisfied. The variables appearing in *Term* are *local* to the *bagof* predicate and must not appear elsewhere in a clause or a query containing *bagof*. If there are free variables in *Goal* not appearing in *Term*, *bagof* can be resatisfied generating alternative values for *List* corresponding to different instantiations of the free variables in *Goal* that do not occur in *Term*. The free variables in *Goal* not appearing in *Term* become therefore grouping variables. By using existential quantification, one can force a variable in *Goal* that does not appear in *Term* to be treated as local.

Let us look at some examples of grouping using the *bagof* predicate.

Example 7.1 Consider program P containing the following facts:

```
p(a,1).
p(a,2).
p(b,3).
p(b,4).
```

Now consider the following query:

```
Q: bagof(Y,p(Z,Y),X).
A: X = [1,2]
```

```
Z = a ? ;
X = [3,4]
Z = b ? ;
no
```

Because `Z` is an uninstantiated free variable, `bagof` treats `Z` as a grouping variable and `Y` as a local variable. Then, for each ground instance of `Z`, such that there exists a value of `Y` such that `p(Z,Y)` holds, `bagof` returns a list `X` containing all instances of `Y`. In this case `bagof` returns two lists: the first containing all instances of `Y` such that `p(a,Y)` holds, the second containing all instances of `Y` such that `p(b,Y)` holds.

In the query above `Y` is a local variable. If we also want to make `Z` local, then we have to explicitly use existential quantification for `Z`:

```
Q: bagof(Y,Z^p(Z,Y),X).
A: X = [1,2,3,4] ? ;
   no
```

Now both `Y` and `Z` are local: `Y` because it appears in *Term*, `Z` because it is explicitly existentially quantified.

In TuLiP, we use modes to guide the credential distribution and discovery and to guarantee groundness of the computed answer substitutions for the queries. Because we want to state the groundness and termination results also for the programs containing grouping atoms, we need a moded version of *bagof*. Therefore we introduce *bagof_m*, which a syntactical variant of *bagof* and is moded. We decided to use a slightly different syntax for *bagof_m* comparing to that of the original *bagof* built-in. First of all we want to make grouping variables explicit in the notation. Secondly, we want to eliminate the need of using the existential quantification for making some of the variables local in the grouping atom. By using different notation we can simplify the definition of local variables in the grouping atom which makes the presentation easier to follow.

Definition 7.3.1 *A* grouping atom *bagof_m is an atom of the form:*

$$A = \mathit{bagof\_m}(t, gl, \mathit{Goal}, x)$$

where t is a term, gl is a list of distinct variables each of which appears in Goal, Goal is an atomic query (but not a grouping atom itself), and x is a free variable.

The *bagof_m* grouping atom has the same semantics as *bagof*, with one exception: the original *bagof* fails if *Goal* has no solution while *bagof_m* returns an empty list (in other words *bagof_m* never fails).

Definition 7.3.1 requires that *Goal* is atomic. This simplifies the treatment (in particular the treatment of modes) and is not a real restriction, as one can always define new predicates to break down a nested grouping atom into a number of grouping atoms that satisfy Definition 7.3.1.

Example 7.2 Consider again the program from Example 7.1. The *bagof_m* equivalent for the query `bagof(Y,p(Z,Y),X)` is

```
bagof_m(Y,[Z],p(Z,Y),X)
```

and for the query `bagof(Y,Z^p(Z,Y),X)`:

```
bagof_m(Y,[],p(Z,Y),X).
```

7.3.1 Semantics of simple *bagof* queries

A subtle difficulty in providing a reasonable semantics for *bagof_m* is due to the fact that we have to take into consideration the multiplicity of answers. In a typical situation, *bagof_m* will be used to compute e.g. averages, as in the query `bagof_m(W,[Y],p(Y,W),X), average(X,Z)`. To this end, `X` should actually be instantiated to a *multiset* of terms corresponding to the answers of the query `p(Y,W)`. Number of researchers investigated the problem of incorporating sets into a logic programming language (see Related Work for an overview). Here, we follow a more practical approach and we represent a multiset with a Prolog list. The disadvantage of using a list is that it is order-dependent: by permuting the elements of a list one can obtain a different list. In the (natural) implementation, given the query *bagof_m*$(\ldots,\ldots,$ *Goal*$, x)$, the c.a.s. will instantiate x to a list of elements, the order of which is dependent on the order with which the computed answer substitutions to the query *Goal* are computed. This depends in turn on the order of the clauses in the program. This means that we cannot provide the declarative semantics for our *bagof_m* construct unless we introduce multisets as first-class citizens of the language.

The fact that we are unable to give fully declarative semantics of *bagof_m* does not prevents us from proving important properties of groundness of the computed answer substitutions and termination of programs containing grouping atoms. In this section, we give two definitions of the computed answer substitution to *bagof_m*: first one – more "declarative" – assumes that multisets of terms are part of the universe of discourse and that a multiset operator $[\![\ ]\!]$ is available, while the second relies on Prolog lists.

Definition 7.3.2 (c.a.s. to *bagof_m* using Multisets) *Let P be a program, and $A =$ bagof_m$(t, gl,$ Goal$, x)$ be a query. The multiset $[\![\alpha_1, \ldots, \alpha_k]\!]$ of computed answer substitutions of $P \cup A$ is defined as follows:*

1. *Let $\Sigma = [\![\sigma_1, \ldots, \sigma_n]\!]$ be the multiset of c.a.s. of $P \cup$ Goal.*
2. *Let $\Sigma_1, \ldots \Sigma_k$ be a partitioning of Σ such that two answers σ_i and σ_j belong to the same partition iff $gl\sigma_i = gl\sigma_j$,*
3. *For each Σ_i, let ts_i be the multiset of terms obtained by instantiating t with the substitutions σ_i in Σ_i, i.e. $ts_i = [\![t\sigma_i \mid \sigma_i \in \Sigma_i]\!]$, and let $gl_i = gl\sigma$ where σ is any substitution from Σ_i.*
4. *For $i \in [1, k]$, α_i is the substitution $\{gl/gl_i, x/ts_i\}$.*

Example 7.3 *Let P be a program containing the following facts: `p(a,c,1)`, `p(a,d,1)`, `p(a,e,3)`, `p(b,c,2)`, `p(b,d,2)`, `p(b,e,4)`. Let $A =$ `bagof_m(Z,[Y],p(Y,W,Z),X)`. Then $P \cup A$ yields the following two computed answer substitutions: $\alpha_1 = \{$`Y`$/a,$ `X`$/[\![1,1,3]\!]\}$ and $\alpha_2 = \{$`Y`$/b,$ `X`$/[\![2,2,4]\!]\}$.*

As we said, since Prolog does not support multisets, in the sequel we use lists instead. The disadvantage of using lists is that they are order-dependent, and that if a multiset contains two or more different elements, then there exists more than one list "representing" it. Here we simply accept this shortcoming and tolerate the fact that, in real Prolog programs, the aggregating variable x will be instantiated to one of the possible lists representing the multiset of answers.

Definition 7.3.3 (c.a.s. to *bagof_m* using Lists) *Let P be a program, and $A =$ bagof_m$(t, gl, Goal, x)$ be a query. The multiset $[\![\alpha_1, \ldots, \alpha_k]\!]$ of computed answer substitution of $P \cup A$ is defined as follows:*

1. *Let $\Sigma = [\![\sigma_1, \ldots, \sigma_n]\!]$ be the multiset of c.a.s. of $P \cup Goal$.*
2. *Let $\Sigma_1, \ldots \Sigma_k$ be a partitioning of Σ such that two answers σ_i and σ_j belong to the same partition iff $gl\sigma_i = gl\sigma_j$,*
3. *For each i, let Δ_i be an ordering on Σ_i, i.e. a list of substitutions containing the same elements of Σ_i, counting multiplicities.*
4. *For each $\Delta_i = [\sigma_{i_1}, \ldots, \sigma_{i_n}]$, let ts_i be the* list *of terms obtained by instantiating t with the substitutions in Δ_i, i.e. $ts_i = [t\sigma_{i_1}, \ldots, t\sigma_{i_n}]$, and let $gl_i = gl\sigma$ where σ is any substitution from Δ_i.*
5. *for $i \in [1, k]$, α_i is the substitution $\{gl/gl_i, x/ts_i\}$.*

Example 7.4 *Take the same program P as in Example 7.3 but with different order of the clauses:* `p(a,c,1)`, `p(a,e,3)`, `p(a,d,1)`, `p(b,e,4)`, `p(b,c,2)`, `p(b,d,2)`. *Again, let $A =$* `bagof_m(Z,[Y],p(Y,W,Z),X)`. *Then $P \cup A$ yields the following two computed answer substitutions using lists: $\alpha_1 = \{$*`Y/a, X/[1,3,1]`*$\}$ and $\alpha_2 = \{$*`Y/b, X/[4,2,2]`*$\}$.*

In the sequel, we refer to this second definition. In order to bring this definition into practice, i.e. to really compute the answer to a query *bagof_m*$(t, gl, Goal, x)$, we have to require that $P \cup$ *Goal* terminates.

7.4 Using *bagof_m* in queries and programs

In this section we discuss the use of *bagof_m* in programs. Here we are going to take advantage of modes, which lets us prove groundness and termination properties.

We begin with the definition of a mode of the *bagof_m* atom.

7.4.1 Modes

The mode of a query *bagof_m*$(t, gl, Goal, x)$ depends on the mode of the *Goal*, so it is not fixed *a priori*. In addition, we introduce the concept of *local variables*.

Definition 7.4.1 *Let $A =$ bagof_m$(t, gl, Goal, x)$. We define the following sets of* input, output *and* local *variables for A:*

- $VarIn(A) = VarIn(Goal)$,
- $VarOut(A) = Var(gl) \setminus VarIn(A) \cup \{x\}$,
- $VarLocal(A) = Var(A) \setminus (VarIn(A) \cup VarOut(A))$,

For example, let $A =$ `bagof_m(q(W,Y,Z),[Y],p(W,Y,Z),X)` be an aggregate atom, and assume that the original mode of `p` is (In, Out, Out). Then, $VarIn(A) = \{$`W`$\}$, $VarOut(A) = \{$`X`, `Y`$\}$, and $VarLocal(A) = \{$`Z`$\}$.

Now, we can extend the definition of well-moded program to take into consideration *bagof_m* atoms; the only extra care we have to take is that local variables should not appear elsewhere in the clause (or query).

Definition 7.4.2 (Well-Moded-Extended) *We say that the clause* $H \longleftarrow B_1, \ldots, B_n$ *is* well-moded *if for all* $i \in [1, n]$

$$VarIn(B_i) \subseteq \bigcup_{j=1}^{i-1} VarOut(B_j) \cup VarIn(H)$$

and

$$VarOut(H) \subseteq \bigcup_{j=1}^{n} VarOut(B_j) \cup VarIn(H).$$

and $\forall B_i \in \{B_1, \ldots, B_n\}$

$$VarLocal(B_i) \cap \left(\bigcup_{j \in \{1,\ldots,i-1,i+1,\ldots,n\}} Var(B_j) \cup Var(H) \right) = \emptyset.$$

A query $\mathbf{A}$ *is* well-moded *iff the clause* $H \longleftarrow \mathbf{A}$ *is well-moded, where* H *is any (dummy) atom of zero arity. A program is* well-moded *if all of its clauses are well-moded.*

7.4.2 LD Derivations with Grouping

We extend the definition of LD-resolution to queries containing *bagof_m* atoms.

Definition 7.4.3 (LD-resolvent with grouping) *Let* P *be a program. Let* $\rho : B, \mathbf{C}$ *be a query. We distinguish two cases:*

1. *if* B *is a bagof_m atom and* α *is a c.a.s. for* B *in* P *then we say that* $B, \mathbf{C}$ *and* P *yield the resolvent* $\mathbf{C}\alpha$*. The corresponding derivation step is denoted by* $B, \mathbf{C} \stackrel{\alpha}{\Longrightarrow}_P \mathbf{C}\alpha$*.*
2. *if* B *is a regular atom and* $c : H \leftarrow \mathbf{B}$ *is a clause in* P *renamed apart wrt* ρ *such that* H *and* B *unify with* mgu θ*, then we say that* ρ *and* c *yield resolvent* $(\mathbf{B}, \mathbf{C})\theta$*. The corresponding derivation step is denoted by* $B, \mathbf{C} \stackrel{\theta}{\Longrightarrow}_c (\mathbf{B}, \mathbf{C})\theta$*.*

As usual, a maximal sequence of derivation steps starting from query $\mathbf{B}$ *is called an* LD derivation of $P \cup \{\mathbf{B}\}$ *provided that for every step the standardisation apart condition holds.* □

Example 7.5 The Financial Administration (`fa`) of the University of Twente makes monthly summaries of the expenses made within several projects. Each expense is represented by a predicate `expense`/4, moded (In, Out, Out, Out), where the first argument is the research group making the expense, the second argument represents the project to be charged, the third argument is the amount used, and the last one is a time-stamp. A research group within a department is denoted by `research_group(Dept,RGroup)` moded (In, Out).

```
expense(dies,ishare,2200,'25-01-2007').
expense(dies,ishare,2200,'25-02-2007').
expense(caes,ishare,1000,'10-03-2007').
expense(caes,ishare,2200,'25-03-2007').
expense(dies,istrice,1200,'25-01-2007').
expense(caes,istrice,1400,'25-02-2007').
research_group(ewi,dies).
research_group(ewi,caes).
```

Now imagine that one is interested in the list of expenses made by each research group in the `ewi` department grouped by the project and formatted as `expense(RGroup,Project,Amount)`. Then one can use the following query:

$$A = \texttt{research\_group(ewi,W)},\ \texttt{bagof\_m(expense(W,Y,Z),[Y],expense(W,Y,Z,V),X)}.$$

We have $VarIn(A) = \{\texttt{W}\}$, $VarOut(A) = \{\texttt{X},\texttt{Y}\}$, and $VarLocal(A) = \{\texttt{V},\texttt{Z}\}$. The computed ground answers to this query are:

```
(1) research_group(ewi,dies),bagof_m(expense(dies,ishare,Z),
        [ishare],expense(dies,ishare,Z,V),
        [expense(dies,ishare,2200),expense(dies,ishare,2200)])
(2) research_group(ewi,dies),bagof_m(expense(dies,istrice,Z),
        [istrice],expense(dies,istrice,Z,V),
        [expense(dies,istrice,1200)])
(3) research_group(ewi,caes),bagof_m(expense(caes,ishare,Z),
        [ishare],expense(caes,ishare,Z,V),
        [expense(caes,ishare,1000),expense(caes,ishare,2200)])
(4) research_group(ewi,caes),bagof_m(expense(caes,istrice,Z),
        [istrice],expense(caes,istrice,Z,V),
        [expense(caes,istrice,1400)])
```

In order to compute the sum and average of the expenses made by a research group grouped by the project, one may extend the program above with the following rules:

```
sum_avg(RGroup,Proj,Sum,Avg,M) :-
                  bagof_m(Z,[Proj],expense(RGroup,Proj,Z,Y),X),
                  sum(X,Sum,Len), Avg is Sum/Len.
sum([],0,0).
sum([H|T],Sum,Len) :- sum(T,Sum1,Len1), Sum is Sum1+H,
                      Len is Len1+1.
```

Example 7.6 *The query A from Example 7.5 has the following bagof representation:*

```
A = research_group(ewi,W),
        bagof(H,V^Z^(H=expense(W,Y,Z),expense(W,Y,Z,V)),X).
```

7.5 Properties

There are two main properties we can prove for programs containing grouping atoms: groundness of computed answer substitutions and – under additional constraints – termination.

Groundness

Well-moded *bagof_m* atoms enjoy the same features as regular well-moded atoms. The following lemma is a natural consequence of Lemma 4.2.3.

Lemma 7.5.1 *Let P be a well-moded program and $A = bagof\_m(t, gl, Goal, x)$ be a grouping atom in which gl is a list of variables. Take any ground σ such that $Dom(\sigma) = VarIn(A)$. Then each c.a.s. θ of $P \cup A\sigma$ is ground on A's output variables, i.e. $Dom(\theta) = VarOut(A)$ and $Ran(\theta) = \emptyset$.*

Proof. By noticing that $VarIn(A) = VarIn(Goal)$ and that each variable in the grouping list *gl* appears in *Goal*, the proof is a straightforward consequence of Lemma 4.2.3.

Termination

Termination is particularly important in the context of grouping queries, because if *Goal* does not terminate (i.e. if some LD derivation starting in *Goal* is infinite) then the grouping atom $bagof\_m(t, gl, Goal, x)$ does not return any answer (it loops).

A concept we need in the sequel is that of *terminating* program; since we are dealing with well-moded programs, the natural definition we refer to is that of *well-terminating* programs.

Definition 7.5.2 *A well-moded program is called* well-terminating *iff all its LD-derivations starting in a well-moded query are finite.*

Termination of (well-moded) logic programs has been exhaustively studied by several authors [13, 14, 23, 31, 45, 82]. Here we follow the approach of Etalle, Bossi, and Cocco [45].

If the grouping atom is only in the top-level query and there are no grouping atoms in the bodies of the program clauses then, to ensure termination, it is sufficient to require that P be well-terminating in the way described by Etalle et al. [45]: i.e. that for every well-moded non grouping atom A, all LD derivations of $P \cup A$ are finite. If this condition is satisfied then all LD derivations of $P \cup Goal$ are finite and then the query $bagof\_m(t, gl, Goal, x)$ terminates (provided it is well-moded).

On the other hand, if we allow grouping atoms in the body of the clauses, then we have to make sure that the program does not include recursion through a grouping atom. The following example shows what can go wrong here.

Example 7.7 Consider the following program:

```
(1) p(X,Z) :- bagof_m(Y,[X],q(X,Y),Z).
(2) q(X,Z) :- bagof_m(Y,[X],p(X,Y),Z).
(3) q(a,1).
(4) q(a,2).
(5) q(b,3).
(6) q(b,4).
```

Here `p` and `q` are defined in terms of each other through the grouping operation. Therefore `p(X,Z)` cannot terminate until `q(X,Y)` terminates (clause 1). Computation of `q(X,Y)` in turn depends on the termination of the grouping operation on `p(X,Y)` (clause 2). Intuitively, one would expect that the model of this program contains `q(a,1)`, `q(a,2)`, `q(b,3)`, and `q(b,4)`. However, if we apply the extended LD resolvent (Definition 7.4.3) to compute the c.a.s. of `p(X,Y)` we see that the computation loops.

In order to prevent this kind of problems, to guarantee termination we require programs to be *aggregate stratified* [60]. *Aggregate stratification* is similar to the concept of *stratified negation* [9, 95], and puts syntactical restrictions on the aggregate programs so that recursion through *bagof_m* does not occur. For the notation, we follow Apt et al. in [9]. Before we proceed to the definition of stratified programs we need to formalise the following notions. Given a program P and a clause $H \longleftarrow \ldots, B, \ldots. \in P$:

- if B is a grouping atom *bagof_m*$(t, gl, Goal, x)$ then we say that $Pred(H)$ *refers to* $Pred($*Goal*$)$;
- otherwise, we say that $Pred(H)$ *refers to* $Pred(B)$.

We say that relation symbol p *depends on* relation symbol q in P, denoted $p \sqsupseteq q$, iff (p, q) is in the reflexive and transitive closure of the relation *refers to*. Given a non-grouping atom B, the definition of B is the subset of P consisting of *all* clauses with a formula on the left side whose relation symbol is $Pred(B)$. Finally, $p \simeq q \equiv p \sqsubseteq q \wedge p \sqsupseteq q$ means that p and q are mutually recursive, and $p \sqsupset q \equiv p \sqsupseteq q \wedge p \not\simeq q$ means that p calls q as a subprogram. Notice that $\sqsupset$ is a well-founded ordering.

Definition 7.5.3 *A program P is called* stratified *if for every clause $H \longleftarrow B_1, \ldots, B_m$, in it, and every B_j in its body we have that*

- *if B_j is a grouping atom $B_j =$ bagof_m$(\ldots, \ldots, Goal, \ldots)$ then $Pred($Goal$) \not\simeq Pred(H)$.*

Given the finiteness of programs it is easy to show that a program P is stratified iff there exists a partition of it $P = P_1 \cup \cdots \cup P_n$ such that for every $i \in [1, \ldots, n]$, and every clause $cl = H \longleftarrow B_1 \ldots, B_m \in P_i$, and every B_j in its body, the following conditions hold:

1. if $B_j =$ *bagof_m*$(\ldots, \ldots, Goal, \ldots)$ then the definition of $Pred($*Goal*$)$ is contained within $\bigcup_{j<i} P_j$,
2. otherwise the definition of $Pred(B)$ is contained within $\bigcup_{j \leq i} P_j$.

Stratification alone does not guarantee termination. The following (obvious) example demonstrates this.

Example 7.8 *Take the following program:*

```
q(X,Y) :- r(X,Y).
r(X,Y) :- q(X,Y).
p(Y,X) :- bagof_m(Z,[Y],q(Y,Z),X).
```

Notice that $q \simeq r$. *This program is (aggregate) stratified, but the query* `p(Y,X)` *will not terminate.*

In order to handle the problem of Example 7.8 we need to modify slightly the classical definition of termination. The following definition relies on the fact that the programs we are referring to are stratified.

Definition 7.5.4 (Termination of Aggregate Stratified Programs) *Let* P *be an aggregate stratified program. We say that* P *is well-terminating if for every well-moded atom* A *the following conditions hold:*

1. *All LD derivations of* $P \cup A$ *are finite,*
2. *For each LD derivation* δ *of* $P \cup A$, *for each grouping atom bagof_m*$(t, gl, Goal, x)$ *selected in* δ, $P \cup$ *Goal terminates.*

The classical definition of termination considers only point (1). Here however, we have grouping atoms which actually trigger a side goal which is not taken into account by (1) alone. This is the reason why we need (2) as well. Notice that the notion is well-defined thanks to the fact that programs are stratified.

To guarantee termination, we can combine the notion of stratified program above with the notion of well-acceptable program introduced by Etalle, Bossi, and Cocco in [45] (other approaches are also possible). We now show how.

Definition 7.5.5 *Let* P *be a program and let* $\mathbf{B}_P$ *be the corresponding Herbrand base. A function* $|\ |$ *is a* moded level mapping *iff*

1. *it is a* level mapping *for* P, *namely it is a function* $|\ | : \mathbf{B}_P \rightarrow \mathbb{N}$, *from ground atoms to natural numbers;*
2. *if* $p(\mathbf{t})$ *and* $p(\mathbf{s})$ *coincide in the input positions then* $|p(\mathbf{t})| = |p(\mathbf{s})|$.

For $A \in \mathbf{B}_P$, $|A|$ *is called the* level *of* A. □

Condition (2) above states that the level of an atom is independent from the terms filling in its output positions. Finally, we can report the key concept we use in order to prove well-termination.

Definition 7.5.6 (Weakly- and Well-Acceptable [45]) *Let* P *be a program,* $|\ |$ *be a level mapping and* M *a model of* P.

- *A clause of P is called* weakly acceptable *(wrt $|\ |$ and M) iff for every ground instance of it, $H \longleftarrow \mathbf{A}, B, \mathbf{C}$,*

$$\text{if} \quad M \models \mathbf{A} \quad \text{and} \quad Pred(H) \simeq Pred(B) \quad \text{then} \quad |H| > |B|.$$

  P is called weakly acceptable with respect to $|\ |$ and M *iff all its clauses are.*

- *A program P is called* well-acceptable *wrt $|\ |$ and M iff $|\ |$ is a* moded *level mapping, M is a model of P and P is weakly acceptable wrt them.* □

Notice that a fact is always both weakly acceptable and well-acceptable; furthermore if M_P is the least Herbrand model of P, and P is well-acceptable wrt $|\ |$ and some model I then, by the minimality of M_P, P is well-acceptable wrt $|\ |$ and M_P as well. Given a program P and a clause $H \longleftarrow \ldots, B, \ldots$ in P, we say that B is *relevant* iff $Pred(H) \simeq Pred(B)$. For the weakly and well-acceptable programs the norm has to be checked only for the relevant atoms, because only the relevant atoms might provide recursion. Notice then that, because we additionally require that programs are stratified, grouping atoms in a clause are not relevant (called as subprograms).
We can now state the main result of this section.

Theorem 7.5.7 *Let P be a well-moded aggregate stratified program.*

- *If P is well-acceptable then P is well-terminating.*

Proof. (Sketch). Given a well-moded atom A, we have to prove that (a) all LD derivations starting in A are finite and that (b) for each LD derivation δ of $P \cup A$, for each grouping atom *bagof_m*$(t, gl, Goal, x)$ selected in δ, $P \cup Goal$ terminates.

To prove (a) one can proceed exactly as done in [45], where the authors use the same notions of well-acceptable program: the fact that here we use a modified version of LD-derivation has no influence on this point: since grouping atoms are resolved by removing them, they cannot add anything to the length of an LD derivation.

On the other hand, to prove (b) one proceeds by induction on the strata of P. Notice that at the moment that the grouping atom is selected, *Goal* is well-moded (i.e., ground in its input position). Now, for the base case if *Goal* is defined in P_1, then, by (a) we have that all LD-derivations starting in *Goal* are finite, and since we are in stratum P_1 (where clause bodies cannot contain grouping atoms) no grouping atom is ever selected in an LD derivation starting in *Goal*. So $P \cup Goal$ terminates.

The inductive case is similar: if *Goal* is defined in P_{i+1}, then, by (a) we have that all LD-derivations starting in *Goal* are finite, and since we are in stratum P_{i+1} if a grouping atom *bagof_m*$(t', gl', Goal', x')$ is selected in an LD derivation starting in *Goal*, we have that $Goal'$ must be defined in $P_1 \cup \cdots \cup P_i$, so that – by inductive hypothesis – we know that $P \cup Goal'$ terminates. Hence the thesis. □

7.6 Related Work

Aggregate and grouping operations are given lots of attention in the logic programming community. In the resulting work we can distinguish two approaches: (1) in which the grouping

and aggregation is performed at the same time, and (2) - which is closer to our approach - in which grouping is performed first returning a multiset and then an aggregation function is applied to this multiset.

In the first approach an *aggregate subgoal* has the following form:

$$group\_by(p(\mathbf{x},\mathbf{z}),[\mathbf{x}],y=\mathbb{F}(E(\mathbf{x},\mathbf{z}))).$$

This is equivalent to:

$$y=\mathbb{F}(\llbracket E(\mathbf{x},\mathbf{z}):\exists(\mathbf{z})p(\mathbf{x},\mathbf{z})\rrbracket).$$

Here $\mathbf{x}$ are the grouping variables, $p(\mathbf{x},\mathbf{z})$ is a so called aggregation predicate, and $E(\mathbf{x},\mathbf{z})$ is a tuple of terms involving some subset of the variables $\mathbf{x}\cup\mathbf{z}$. $\mathbb{F}$ is an aggregate function that maps a multiset to a single value. The variables $\mathbf{x}$ and y are free in the subgoal while $\mathbf{z}$ are local and cannot appear outside the aggregate subgoal. In other words, if a variable does not appear on the grouping list, this variable is local. For instance, given a program $P=\{$`p(a,1),p(a,2),p(b,3),p(b,4)`$\}$, the query `group_by(p(X,Z),[],Y=sum(Z))` returns the answer `Y=10` whereas the query `group_by(p(X,Z),[X],Y=sum(Z))` returns two answers: `X=a, Y=3` and `X=b,Y=7`. In the former case `X` and `Z` are both local variables while in the latter case only `Z` is local. By requiring that each variable appearing in the aggregation predicate but not appearing on the grouping list to be local, $group\_by$ is slightly more restrictive than *bagof* in this respect. Consider the following query:

```
q(Y), bagof(Z,p(Y,Z),X), sum(X,W).
```

and its *bagof_m* equivalent (assuming that *mode*$(p/2)=(In,\ Out)$ and *mode*$(q/1)=(Out)$):

```
q(Y), bagof_m(Z,p(Y,Z),[Y],X), sum(X,W).
```

This cannot be done with $group\_by$ because the query:

```
q(Y), group_by(p(Y,Z),[],W=sum(Z)).
```

is not a valid one because `Y` is a local variable and cannot appear outside the $group\_by$ atom.

The early declarative semantics for $group\_by$ was given by Mumick et al. [73]. In this work, aggregate stratification is used to prevent recursion through aggregates. Later, Kemp and Stuckey [60] provide the declarative semantics for $group\_by$ in terms of well-founded and stable semantics. They also examine different classes of aggregate programs: aggregate stratified, group stratified, magical stratified, and also monotonic and semi-ring programs. From a more recent work, Faber et al. [47] also relay on aggregate stratification and they define a declarative semantics for disjunctive programs with aggregates. They use the intentional set definition notation to specify the multiset for the aggregate function. Denecker et al. [39] point out that requiring the programs to be aggregate stratified might be too restrictive in some cases and they propose a stronger extension of the well-founded and stable model semantics for logic programs with aggregates (called *ultimate* well-founded and stable semantics). In their approach, Denecker et al. use the Approximation Theory [38]. The work of Denecker et al. is continued and further extended by Pelov et al. [81].

In the second approach, where the grouping is separated from aggregation (as in our approach), the grouping operation is represented by an intentional set definition. This approach

uses an (intentional) set construction operator returning a multiset of answers which is then passed as an argument of an aggregate function:

$$m = [\![E(\mathbf{x}, \mathbf{z}) : \exists(\mathbf{z}) p(\mathbf{x}, \mathbf{z})]\!], y = \mathbb{F}(m).$$

To be handled correctly (with a well defined declarative semantics), this approach requires multisets to be introduced as first-class citizens of the language.

Dovier, Pontelli, and Rossi [41] introduce intentionally defined sets into the constraint logic programming language $CLP(\{\mathcal{D}\})$ where $\mathcal{D}$ can be for instance $\mathbb{FD}$ for finite domains or $\mathbb{R}$ for real numbers. In their work, Dovier et al. concentrate on the set-based operations and so, they do not consider multisets directly. Interestingly, they treat the intentional set definition as a special case of an aggregate subgoal in which $\mathbb{F}$ is a function which given a multiset m as an argument returns the set of all elements in m - i.e. $\mathbb{F}$ removes duplicates from m.

Introducing (multi)sets to a pure logic programming language (i.e. not relaying on a CLP scheme) is also a well-researched area. From the most prominent proposals, Dovier et al. [40] propose an extended logic programming language called $\{\log\}$ (read "set-log") in which sets are first-class citizens. The authors introduce the basic set operations like set membership $\in$ and set equality $=$ along with their negative counterparts $\notin$ and $\neq$. Also, in the related work, Dovier et al. [40] show a nice overview of other logic programming systems incorporating set primitives. Concerning multisets directly, Ciancarini et al. [35] show how to extend a logic programming language with multisets. They strictly follow the approach of Dovier et al. [40].

Important to notice here, is that these earlier works of Dovier et al. and Ciancarini et al. (as well as most of other related work on embedding sets in a logic programming language - see Dovier et al. [40, 41] for examples) focus on the so called *extensional set construction* - which basically means that a set is constructed by enumerating the elements of the set. This is not suitable for our work as this does not enables us to perform grouping.

Moded Logic Programming is well-researched area [12, 15, 71, 93]. However, modes have been never applied to aggregates. We also extend the standard definition of a mode to include the notion of *local variables*. By incorporating the mode system we are able to state the groundness and termination results for the *bagof*-like operations.

7.7 Conclusions

In this chapter we study the grouping operations in Prolog using the standard Prolog built-in predicate *bagof*. Grouping is needed if we want to perform aggregation, and we need aggregation in TuLiP to be able to model reputation systems. In order to make the grouping operations easier to integrate with TuLiP, we add modes to *bagof* (we call the moded version *bagof_m*). We extend the definition of a mode by allowing some variables in a grouping atom to be *local*. Finally, we show that for the class of well-terminating aggregate stratified programs the basic properties of well-modedness and well-termination also hold for programs with grouping.

CHAPTER 8

Conclusions and Future Work

We distinguish two approaches towards trust management: reputation based and credential based trust management. In reputation based trust management a security decision is based on past user behaviour provided in terms of the *feedback* from other users. Feedback is usually a subjective opinion of a user about another user based on the previous interaction between those two users. In reputation based trust management feedback is also called a *recommendation*. Because of its subjective nature, a recommendation is rarely a binary *yes* or *no*, but the feedback can have values from an arbitrary continuous domain. In credential based trust management a security decision is based on security credentials. Every user can issue a credential that can then be used by other users. There is no fussiness in answering an authorisation request: the answer is either *yes* or *no* (or sometimes *don't know*).

In this thesis we study credential based distributed trust management. In (credential based) distributed trust management, the credentials are not only issued by different users, but can also be stored in a distributed way: a single user may store her own credentials, but also the credentials issued by other users. In distributed trust management there is no central repository, where the users should store their credentials. A successful distributed trust management system should not only allow the users to write the credentials, but also help the user to distribute the credentials in such a way that the credentials can be found later for the compliance checking. Therefore, a distributed trust management system should satisfy the following requirements:

1. a simple yet expressive trust management language,
2. a compliance checker,
3. support for the distributed credential storage.

Most of the existing trust management systems do not satisfy all of these requirements at the same time. For instance, the PeerTrust trust management system [74], provides an expressive trust management language and a compliance checker capable of evaluating the credentials but does not support credential discovery. On the other hand, the RT family of trust management languages [66] provides a storage type system to support distributed credential storage, but the trust management language is complex when sophisticated policies need to be mod-

elled. Our aim therefore is to design a generic open-ended distributed trust management system that can satisfy all the above stated requirements.

In this thesis we achieve this aim successfully. Our contributions, which all together lead us to this positive conclusion, are the following:

1. We introduce a new member to the RT family of trust management languages: $RT_{\ominus}$. The RT framework is a family of trust management languages with gradually increasing expressive power. Each language from the RT family is monotonic, which means that RT does not support negation. We show in Chapter 3 that some policies (likes separation of duty) can be expressed naturally only if we admit at least a restricted form of negation to the language. RT tries to fix the problem in the more expressive members of the family by introducing the so called "manifold roles". A manifold role can contain not only entities but also collections of entities. Unfortunately, the use of manifold roles makes the language harder to understand and it becomes difficult to model more sophisticated scenarios such as thresholds or separation of duty. $RT_{\ominus}$ supports a restricted form of negation (we call it "negation in context") by means of a new operator $\ominus$. The restricted form of negation in $RT_{\ominus}$ is sufficient to model sophisticated, realistic policies like thresholds or separation of duty in a natural way without all the complexity brought by an unlimited form negation.

2. We design a distributed trust management system TuLiP. A disadvantage of RT (so also of $RT_{\ominus}$) is that the syntax and the semantics are not uniform across different members of the family, which makes the syntax and the semantics complex with the more expressive members. Because our goal is to have a trust management systems which provides a simple yet expressive language for writing credentials, instead of patching RT by adding new members, which would result in a more complex language, we start from scratch and build our own trust management system TuLiP. TuLiP has a powerful trust management language based on logic programming, and offers the same uniform syntax regardless of the complexity of the policy modelled. Another disadvantage of RT is that the storage type system is not fully reflected in the semantics of the language. By contrast, in TuLiP it is impossible to write a credential without thinking about the credential storage. In our approach the mode associated with a credential atom indicates where the corresponding credential should be stored so that it can be found later when needed. The storage information is therefore an integral part of our trust management language. Finally, TuLiP comes with a compliance checker, LIAR, which, besides doing a typical compliance checker reasoning job, also discovers and fetches the credentials in a goal oriented way using the mode information. We prove that LIAR is sound and complete w.r.t. to the declarative semantics of TuLiP.

3. We show how to implement and deploy TuLiP on a distributed system. We provide a proof of concept implementation consisting of LIAR, a number of credential servers, a mode register, and the user interface. In our implementation we show that TuLiP can be realised using of-the-shelf technology with minimal requirements on the underlying infrastructure. Our system does not depend on a centralised authority (the mode register is centralised at the moment but it is not required when doing credential discovery) and it does not require a heavy public key infrastructure (PKI). We also provide a demo showing the concrete application of TuLiP in a distributed P2P content management system.

We learnt many lessons while working on the thesis. Here we mention only the most important ones. On the theoretical side, we learnt how difficult is to add non-monotonic features to the language. Most of the existing trust management languages avoid non-monotonic extensions (like negation) in order to avoid all the complexity brought by these extensions. There is a number of available semantics for logic programming with negation but it is not trivial to decide which is the right one for the distributed trust management. When working on non-monotonic extensions in Chapter 3, we learnt that giving a binary "*yes*" or "*no*" answer to an authorisation query is not always possible - sometimes we have to deal with simple "*I don't know*". Proving soundness and completeness of LIAR shown us how important it is to carefully consider the range of features that the system should provide. For instance, adding an extensive support for subject-traceable credential chains resulted in a higher complexity of the operational semantics (and a more difficult proof of soundness and completeness of LIAR). We believe that the operational semantics could be simplified if one accepts a restricted support for subject traceable credentials.

Finally, the practical work was inspiring and allowed us to identify the problems we had in the theory which would be hard to spot without touching the "real thing". This is especially true for the LIAR algorithm, for which it took several iterations before having fully operational version which could be proven sound and complete. In our practical work we used several programming languages and technologies like Prolog, Python, Action Script (Flash), PHP, HTML, or CSS. Our implementation amounts to approximately 10000 lines of source code including many comments and the repeated code, which means that a more professional (commercial) implementation can be build quickly by a small team.

Future Directions TuLiP is not yet complete and there is still work to be done. The most important extension is incorporating support for reputation based trust management into TuLiP. To do this, one needs to add support for grouping and aggregation to the trust management language as these are important elements of each reputation system. In Chapter 7 we show how to handle grouping and aggregation in logic programming in such a way that important properties of groundness and termination are preserved. By showing this, we open the door for the future research of how to integrate grouping and aggregation into TuLiP and to which extent one can model the scenarios from the world of reputation systems in TuLiP. The Integration of Reputation Systems and credential based Trust Management is not the only way the TuLiP trust management system can be improved or extended. Below we enumerate the most important directions:

1. The first important research direction is supporting non-monotonic features in TuLiP as pioneered in $RT_{\ominus}$. This research should concentrate on the following two aspects: (1) providing appropriate declarative semantics, (2) extending the storage type system, and (3) updating the LIAR algorithm. One can further distinguish between using stratified and non-stratified programs. The former simplifies the definition of the declarative semantics and also is aligned with the semantics of programs with grouping which we show in Chapter 7. On the other hand, having non-stratified programs may actually simplify the design of the LIAR algorithm that in such a case does not have to deal with the distributed stratification. An interesting challenge would be also to design the storage type system for $RT_{\ominus}$.

2. Another interesting research direction is improving the privacy of the users in TuLiP.

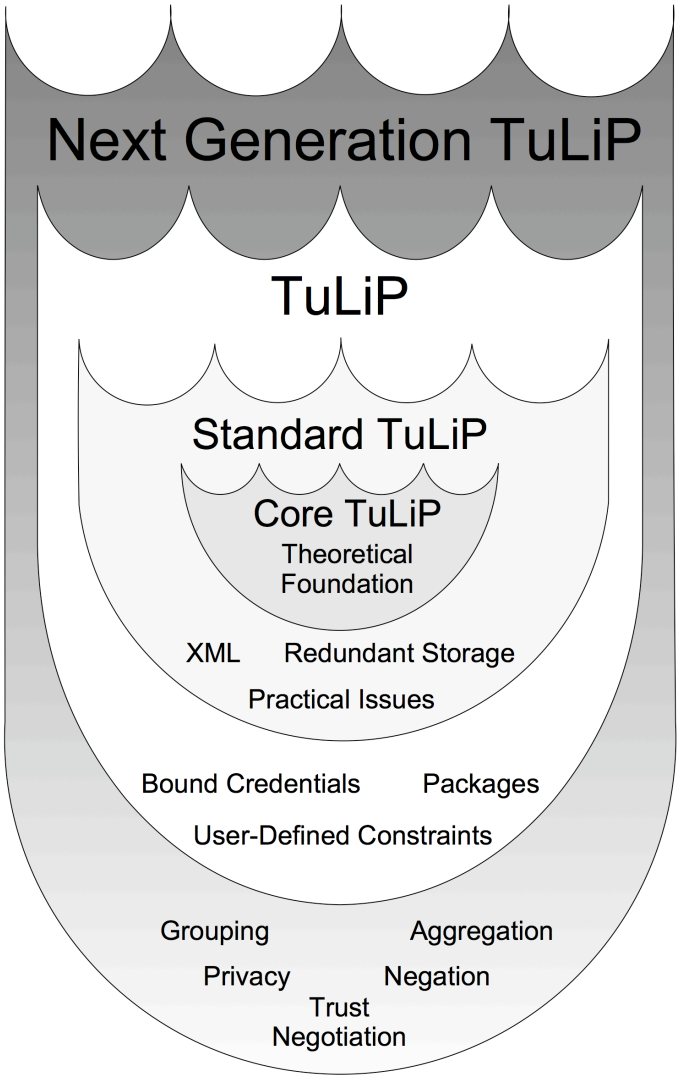


Fig. 8.1: Next Generation (NG) TuLiP

Here, the user privacy may be improved if we avoid unnecessary exchange of the credentials and instead only send queries and receive answers. This leads to a more distributed computational model, where many entities actively participate in answering the query - each one doing only a small amount of work. The research should also answer the question when such a scenario is applicable (in other words, when we do have to send the credentials and when we can avoid it). Finally, one may need to investigate how to build a distributed version of the LIAR algorithm which can handle cycles and stratification correctly.

3. TuLiP can be used for Trust Negotiation by treating credentials as ordinary resources Here one may want to have an explicit support for negotiation policies, which allow us to describe when and which credentials can be revealed in the same way as we define standard credentials.

4. TuLiP is close to becoming a real world system. It still requires several improvements at the implementation level. Improving the implementation of the TuLiP trust management system may give new insight into what is still missing or what is redundant. This, in turn, may lead to new research questions. Currently, we see the need to investigate the following implementation-level issues: (a) consistency of the mode assignment (b) supporting multiple credential formats, (c) encryption, helper tools, graphical user interfaces, (d) plugable infrastructure - open architecture (going open source?).

Figure 8.1 draws the features of the Next Generation TuLiP on the canvas of the work already carried out.

Bibliography

Author References

[1] M. Czenko, H. Tran, J. Doumen, S. Etalle, P. Hartel, and J. den Hartog. Nonmonotonic Trust Management for P2P Applications. In *Proc. 1st International Workshop on Security and Trust Management*, Electronic Notes in Theoretical Computer Science (ENTCS), pages 101–116. Elsevier, 2005.

[2] M. R. Czenko, J. M. Doumen, and S. Etalle. Trust Management in P2P Systems Using Standard TuLiP. In *Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security, Trondheim, Norway*, volume 263/2008 of *IFIP International Federation for Information Processing*, pages 1–16, Boston, May 2008. Springer.

[3] M. R. Czenko and S. Etalle. Logic Programming with Grouping Made Easy Using Modes. To be submitted.

[4] M. R. Czenko and S. Etalle. Core TuLiP - Logic Programming for Trust Management. In *Proc. 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal*, volume 4670 of *LNCS*, pages 380–394, Berlin, 2007. Springer Verlag.

[5] M. R. Czenko, S. Etalle, D. Li, and W. H. Winsborough. An Introduction to the Role Based Trust Management Framework RT. In *Foundations of Security Analysis and Design IV – FOSAD 2006/2007 Tutorial Lectures*, volume 4677 of *LNCS*, pages 246–281. Springer Verlag, 2007.

Other References

[6] A. Abdul-Rahman and S. Hailes. Supporting Trust in Virtual Communities. In *Proc. 33rd Hawaii International Conference on System Sciences*, volume 6, page 6007. IEEE Computer Society Press, 2000.

[7] ANSI. American National Standard for Information Technology – Role Based Access Control. ANSI INCITS 359-2004, February 2004.

[8] A. W. Appel and E. W. Felten. Proof-Carrying Authentication. In *CCS '99: Proc. 6th ACM Conference on Computer and Communications Security*, pages 52–62. ACM Press, 1999.

[9] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[10] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[11] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In *Algebraic Methodology and Software Technology (AMAST)*, volume 936 of *LNCS*, pages 66–90. Springer, 1995.

[12] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[13] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[14] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[15] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.

[16] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. on Information and System Security*, 6(4):501–546, 2003.

[17] L. Bauer, M. A. Schneider, and E. W. Felten. A General and Flexible Access-Control System for the Web. In *Proc. 11th USENIX Security Symposium*, pages 93–108. USENIX Association, 2002.

[18] Ljudevit Bauer. *Access control for the web via proof-carrying authorization*. PhD thesis, Princeton, NJ, USA, 2003. Adviser-Andrew W. Appel.

[19] M. Y. Becker and P. Sewell. Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *Proc. 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 159–168. IEEE Computer Society Press, 2004.

[20] M. Y. Becker and P. Sewell. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. In *Computer Security Foundations Workshop (CSFW)*, pages 139–154. IEEE Computer Society Press, 2004.

[21] E. Bertino, E. Ferrari, and A. Squicciarini. $\mathcal{X}$-TNL: An XML-based Language for Trust Negotiations. In *Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 81. IEEE Computer Society, 2003.

[22] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-$\mathcal{X}$: A Peer-to-Peer Framework for Trust Establishment. *IEEE Trans. Knowl. Data Eng.*, 16(7):827–842, 2004.

[23] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.

[24] BitTorrent. *The global standard for delivering high-quality files over the Internet.*, 2007. `http://www.bittorrent.com`.

[25] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System, Version 2. IETF RFC 2704, September 1999.

[26] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems Security. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 185–210. Springer-Verlag, 1999.

[27] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. 17th IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[28] M. Blaze, J. Ioannidis, and A. D. Keromytis. Experience with the KeyNote Trust Management System: Applications and Future Directions. In *iTrust*, volume 2692 of *Lecture Notes in Computer Science*, pages 284–300. Springer, 2003.

[29] G. Boella and L. van der Torre. Permission and Authorization in Policies for Virtual Communities of Agents. In *Proc. Agents and P2P Computing Workshop at AAMAS'04*. Springer Verlag, 2004.

[30] P. Bonatti, C. Duma, D. Olemdilla, and N. Shahmehri. An Integration of Reputation-based and Policy-based Trust Management. In *Proc. Semantic Web and Policy Workshop*, 2005.

[31] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124:297–328, 1994.

[32] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[33] M. Speciner C. Kaufman, R. Perlman. *Network Security: Private Communication in a Public World.* Prentice Hall PTR, 2 edition, April 2002.

[34] W. Chen, T. Swift, and D.S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[35] P. Ciancarini, D. Fogli, and M. Gaspari. A Logic Language based on GAMMA-like Multiset Rewriting. In *Extensions of Logic Programming (ELP)*, volume 1050 of *LNCS*, pages 83–101. Springer, March 1996.

[36] D. Clarke, J.E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[37] C. Dellarocas. Analyzing the Economic Efficiency of eBay-like Online Reputation Reporting Mechanisms. In *EC '01: Proc. 3rd ACM conference on Electronic Commerce*, pages 171–179, New York, NY, USA, 2001. ACM Press.

[38] M. Denecker, V. Marek, and M. Truszczyński. *Approximations, Stable Operators, Well-Founded Operators, Fixpoints and Applications in Nonmonotonic Reasoning*, volume 597 of *The Springer International Series in Engineering and Computer Science*, chapter 6, pages 127–144. Springer, 2001.

[39] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates. In *ICLP*, volume 2237 of *LNCS*, pages 212–226. Springer, 2001.

[40] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A logic programming language with finite sets. In *ICLP*, pages 111–124. MIT Press, 1991.

[41] A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. In *Logic Programming*, volume 2916 of *LNCS*, pages 284–299, Berlin, 2003. Springer.

[42] P.M. Dung and P.M. Thang. Trust Negotiation with Nonmonotonic Access Policies. In *Proc. IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM 04)*, volume 3283 of *LNCS*, pages 70–84. Springer-Verlag, 2004.

[43] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. *Declarative problem-solving using the DLV system*, chapter 4, pages 79–103. Kluwer Academic Publishers, 2000.

[44] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. IETF RFC 2693, September 1999.

[45] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *J. Log. Program.*, 38(2):243–257, 1999.

[46] S. Etalle and W. H. Winsborough. A Posteriori Compliance Control. In *Proc. 12th ACM Symposium on Access Control Models and Technologies, Nice, France*, pages 11–20. ACM Press, 2007.

[47] W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Logics in Artificial Intelligence (JELIA)*, volume 3229 of *LNCS*, pages 200–212. Springer, 2004.

[48] M. Fitting. A Kripke-Kleene semantics for Logic Programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[49] A.V. Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[50] M. Gelfond and V. Lifschitz. The Stable Model Semantics For Logic Programming. In *Proc. 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[51] GnuPG. *http://www.gnupg.org/download/integrity_check.en.html*.

[52] L. Gong, R. Needham, and R. Yahalom. Reasoning About Belief in Cryptographic Protocols. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society Press, 1990.

[53] C. Gunter and T. Jim. Policy-directed Certificate Retrieval. *Software: Practice & Experience*, 30(15):1609–1640, September 2000.

[54] A. Herzberg, Y. Mass, J. Michaeli, Y. Ravid, and D. Naor. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–14. IEEE Computer Society Press, 2000.

[55] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for Multiple Access Control Policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.

[56] S. L. Jarvenpaa, N. Tractinsky, and M. Vitale. Consumer Trust in an Internet Store. *Inf. Tech. and Management*, 1(1-2):45–71, 2000.

[57] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proc. IEEE Symposium on Security and Privacy*, pages 106–115. IEEE Computer Society Press, 2001.

[58] A. Jøsang. The Right Type of Trust for Distributed Systems. In *NSPW '96: Proc. Workshop on New Security Paradigms*, pages 119–131. ACM Press, 1996.

[59] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proc. 12th International Conference on World Wide Web*, pages 640–651. ACM Press, 2003.

[60] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *ISLP*, pages 387–401. MIT Press, 1991.

[61] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4(4):289–308, 1987.

[62] F. Lee, D. Vogel, and M. Limayem. Adoption of informatics to support virtual communities. In *HICSS '02: Proc. 35th Annual Hawaii International Conference on System Sciences -Volume 8*, page 214.2. IEEE Computer Society Press, 2002.

[63] N. Li, J. Feigenbaum, and B. N. Grosof. A Logic-based Knowledge Representation for Authorization with Delegation (Extended Abstract). In *Proc. 1999 IEEE Computer Security Foundations Workshop*, pages 162–174. IEEE Computer Society Press, June 1999.

[64] N. Li, B. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

[65] N. Li and J. Mitchell. RT: A Role-based Trust-management Framework. In IEEE Computer Society Press, editor, *Proc. 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 201–212, 2003.

[66] N. Li, J. Mitchell, and W. Winsborough. Design of a Role-based Trust-management Framework. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.

[67] N. Li, W. Winsborough, and J. Mitchell. Distributed Credential Chain Discovery in Trust Management. *Journal of Computer Security*, 11(1):35–86, 2003.

[68] LIACC/Universidade do Porto and COPPE Sistemas/UFRJ. *YAP Prolog*, April 2006.

[69] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2 edition, 1993.

[70] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS*, volume 2429 of *LNCS*, pages 53–65. Springer, 2002.

[71] C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.

[72] L. Mui, M. Mohtashemi, and A. Halberstadt. A Computational Model of Trust and Reputation for E-businesses. volume 07, page 188. IEEE Computer Society, 2002.

[73] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proc. 16th International Conference on Very Large Databases*, pages 264–277. Morgan Kaufmann Publishers Inc., 1990.

[74] W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web. In *Secure Data Management*, pages 118–132, 2004.

[75] I. Niemel and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 421–430. Springer-Verlag, 1997.

[76] OASIS. *Assertions and Protocols for the OASIS: Security Assertion Markup Language (SAML) V2.0*, March 2005. `http://www.oasis-open.org/committees/documents.php?wg_abbrev=security`.

[77] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0*, Feb 2005. `http://www.oasis.org`.

[78] OASIS. *SAML V2.0 Executive Overview*, April 2005. `http://www.oasis-open.org/committees/documents.php?wg_abbrev=security`.

[79] OASIS. *Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite*, February 2007. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`.

[80] L. Pearlman, V. Welch, I. T. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 50–59. IEEE Computer Society Press, 2002.

[81] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP)*, 7(3):301–353, 2007.

[82] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *LNCS*. Springer, 1990.

[83] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 205–216, 2005.

[84] T. C. Przymusinski. Perfect model semantics. In *Fifth international Conference and Symposium on Logic programming, Seattle, U.S.A.*, pages 1081–1096. MIT Press, 1988.

[85] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 161–172. ACM, 2001.

[86] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, 2000.

[87] R. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure, October 1996. Available at http://theory.lcs.mit.edu/∼rivest/sdsi11.html.

[88] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, pages 329–350, 2001.

[89] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for Policy Languages for Trust Negotiation. In *Proc. 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 68–80. IEEE Computer Society Press, 2002.

[90] D. Seipel. Processing xml-documents in prolog. In *Proc. 17th Workshop on Logic Programming WLP*, 2002.

[91] J. C. Shepherdson. Negation in Logic Programming. In *Foundation of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[92] V. Shmatikov and C. L. Talcott. Reputation-based Trust Management. *Journal of Computer Security*, 13(1):167–190, 2005.

[93] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, 1995. available at http://www.cs.mu.oz.au/mercury/papers.html.

[94] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[95] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[96] W3C. *XML-Signature Syntax and Processing*, Feb 2002. `http://www.w3.org/TR/xmldsig-core/`.

[97] W3C. *Extensible Markup Language (XML) 1.1 (Second Edition)*, Sep 2006. `http://www.w3.org/TR/2006/REC-xml11-20060816/`.

[98] W3C. *Namespaces in XML 1.0 (Second Edition)*, Aug 2006. `http://www.w3.org/TR/REC-xml-names/`.

[99] L. Wang, D. Wijesekera, and S. Jajodia. A Logic-based Framework for Attribute Based Access Control. In *Proc. ACM workshop on Formal methods in security engineering FMSE'04*, pages 45–55. ACM Press, 2004.

[100] S. Weeks. Understanding Trust Management Systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 94–105. IEEE Computer Society Press, 2001.

[101] W. H. Winsborough and N. Li. Towards Practical Automated Trust Negotiation. In *POLICY*, pages 92–103. IEEE Computer Society Press, 2002.

[102] M. Winslett, C. C. Zhang, and P. A. Bonatti. PeerAccess: a logic for distributed authorization. In *ACM Conference on Computer and Communications Security*, pages 168–179. ACM, 2005.

[103] L. Xiong and L. Liu. A Reputation-based Trust Model for Peer-to-Peer eCommerce Communities. In *ACM Conference on Electronic Commerce*, pages 228–229. ACM, 2003.

[104] L. Xiong and L. Liu. PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. *IEEE Trans. Knowl. Data Eng.*, 16(7):843–857, 2004.

[105] R. Yahalom, B. Klein, and T. Beth. Trust Relationships in Secure Systems – A Distributed Authentication Perspective. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, pages 150–164. IEEE Computer Society, 1993.

[106] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[107] P. Zimmermann. *PGP User's Guide*. MIT Press, Cambridge, 1994.

Samenvatting

In de sterk gedistribueerde en heterogene "internet wereld" van vandaag de dag is het delen van resources een dagelijkse activiteit van elke internet gebruiker geworden. We kopen en verkopen spullen over het internet, delen onze vakantiefoto's via facebook™, "tuben" onze thuisvideo's op You Tube™ en wisselen onze interesses en gedachtes uit via blogs. We podcasten, netwerken via Linkedin™, delen bestanden op P2P netwerken en zoeken advies op talrijke online discussie groepen. Hoewel we in de meeste gevallen een zo groot mogelijk groep gebruikers willen bereiken, realiseren we ons vaak dat sommige informatie privé moet blijven, of op zijn minst beperkt tot een zorgvuldig gekozen publiek. Toegangsbeheer is niet langer het domein van de computerbeveiligings expert maar iets wat we elke dag tegenkomen.

In een standaard toegangsbeheer scenario heeft de resource leverancier de volledige controle over de beschermde resource. De beheerder beslist wie toegang heeft tot de resource en welke acties met de resource mogelijk zijn. De groep entiteiten die toegang heeft tot een beschermde resource kan statisch gedefinieerd worden, en is vooraf bekend bij de resource beheerder. Hoewel nog steeds geldig, in veel gevallen is een dergelijk scenario vandaag de dag te beperkend. De toegangbeheerder is niet alleen vereist, maar wil vaak ook de grootst mogelijke groep gebruikers bereiken en veel van deze gebruikers blijven anoniem voor the leverancier. Er is een flexibelere aanpak van toegangsbeheer nodig.

Trust Management is een recente toegangsbeheermethode waarin het autorisatie besluit gebaseerd is op *beveiligingswaarmerken* (credentials). In een waarmerk drukt de *uitgever* (issuer) van het waarmerk attributen (rollen, eigenschappen) van de *ontvanger* (subject) van het waarmerk uit. De waarmerken worden geschreven in een trust management taal om er voor te zorgen dat waarmerken dezelfde betekenis hebben voor alle gebruikers. Een speciaal algoritme, genaamd *compliance checker*, wordt gebruikt om vast te stellen of een set van waarmerken een actie op een beschermde resource toelaat. Een belangrijke eigenschap van trust management is dat iedere entiteit waarmerken mag uitgeven.

In de oorspronkelijke trust management aanpak worden waarmerken op bekende locaties opgeslagen zodat het compliance checker algoritme weet waar te zoeken voor waarmerken. Een andere aanpak is om de gebruikers zelf de waarmerken te laten bewaren. Gedistribueerde opslag van de waarmerken voorkomt de *single point of failure* geïntroduceerd door de centrale waarmerken depot maar vereist dat de het compliance checker algoritme weet waar de waarmerken te vinden zijn. Een ander probleem van de gedistribueerde aanpak is dat het ontwerp van een correct waarmerk ontdekkingsalgoritme beperkingen oplegt aan de uitdrukkingskracht van de trust management taal.

In dit proefschrift laten we zien dat het mogelijk is om een generiek en open trust management system te bouwen dat een sterke uitdrukkingskracht combineert met gedistribueerde waarmerkopslag. Om precies te zijn; we laten zien hoe een trust management systeem te bouwen met:

- een formele maar toch expressieve trust management taal voor het uitdrukken van waarmerken,

- een compliance checker algoritme voor het vaststellen van autorisatie aan de hand een gegeven verzameling waarmerken,
- ondersteuning voor gedistribueerde waarmerkenopslag.

We noemen ons trust management systeem TuLiP (Trust management based on Logic Programming).

In dit proefschrift geven we ook aan hoe TuLiP gebruikt kan worden in een gedistribueerd inhoudsbeheersysteem (we gebruiken foto's als inhoud in onze implementatie). Door gebruik van deze aanpak kan TuLiP bestaande P2P inhoud deling services verbeteren door persoonlijk, schaalbare en wachtwoordvrije toegangscontrole aan de gebruikers te bieden. Door de architectuur te decentraliseren zouden systemen zoals facebook™ of You Tube™ ook kunnen profiteren van TuLiP. Door het eenvoudig te gebruiken en schaalbare toegangscontrole mechanisme, maakt TuLiP het delen mogelijk van persoonlijk en auteursrechtelijk beschermd materiaal via een uniform en bekende gebruikers interface. Hier kan TuLiP business modellen mogelijk maken waarin aanbevolen klanten en klanten van bevriende bedrijven deelnemen in geïndividualiseerde klantenbindingsprogrammas (zoals aantrekkelijke kortingen). Wij zijn overtuigd dat Tulip, door de natuurlijke ondersteuning van samenwerken van autonome entiteiten door middel van gedistribueerde waarmerken, het valideren van business relaties eenvoudiger maakt en hierdoor het ontstaan van nieuwe business modellen bevordert.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed*

Systems: Semantics, Implementation and Composition. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16